27503 CP

PCT

WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : G06F 9/06

A1

(11) International Publication Number: WO 98/09213

(43) International Publication Date: 5 March 1998 (05.03.98)

(21) International Application Number: PCT/US97/15018

(22) International Filing Date: 27 August 1997 (27.08.97)

(30) Priority Data:
08/703,463 27 August 1996 (27.08.96) US

(71) Applicant: VISUAL TELESYSTEMS, INC. [US/US]; Lakeside Office Park, Suite 2, 591 North Avenue, Wakefield, MA 01880 (US).

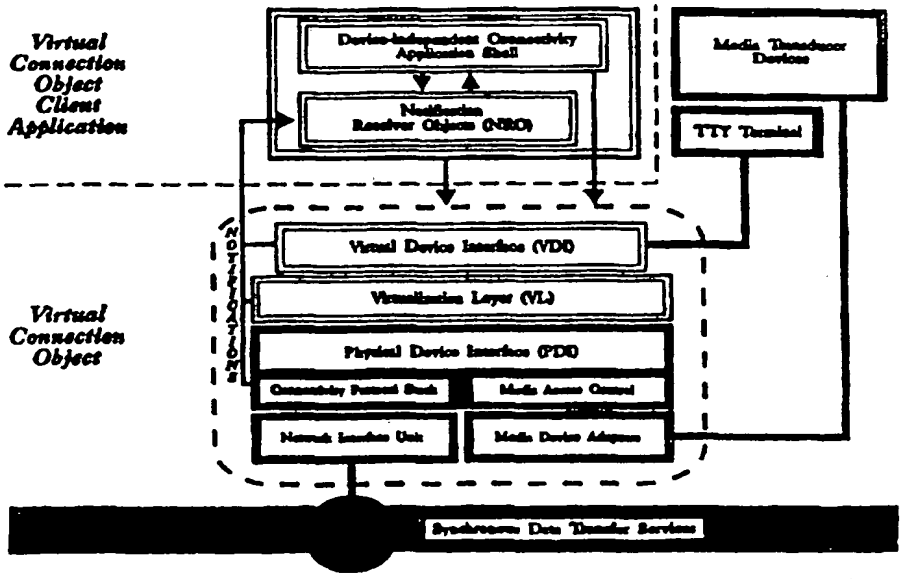
(72) Inventor: GIRARD, Gregory, D.; 106 Hale Street, Beverly, MA 01915 (US).

(74) Agent: PRAHL, Eric, L.; Fish & Richardson, P.C., 225 Franklin Street, Boston, MA 02110-2804 (US).

(81) Designated States: CA, IL, JP, European patent (AT, BE, CH, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

Published
With international search report.
Before the expiration of the time limit for amending the claims and to be republished in the event of the receipt of amendments.

(54) Title: VIRTUALIZED MULTIMEDIA CONNECTION SYSTEM



(57) Abstract

A multimedia connectivity program residing in computer readable memory, the connectivity program when executed on a computer providing to an application program multimedia connectivity services through a real-time multimedia device control subsystem including components selected from among a plurality of multimedia devices and a plurality of real-time multimedia protocol stacks, the program including: a single binary object encapsulating a virtual device interface and a device control interface, the virtual device interface including a plurality of virtual methods that represent logical operations available to the application program for controlling the multimedia device control subsystem, the plurality of virtual functions being completely independent of the components within the device control subsystem, the device control interface mapping the plurality of virtual functions to physical control methods which control the components of the multimedia control subsystem.

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	ES	Spain	LS	Lesotho	SI	Slovenia
AM	Armenia	FI	Finland	LT	Lithuania	SK	Slovakia
AT	Austria	FR	France	LU	Luxembourg	SN	Senegal
AU	Australia	GA	Gabon	LV	Latvia	SZ	Swaziland
AZ	Azerbaijan	GB	United Kingdom	MC	Monaco	TD	Chad
BA	Bosnia and Herzegovina	GE	Georgia	MD	Republic of Moldova	TG	Togo
BB	Barbados	GH	Ghana	MG	Madagascar	TJ	Tajikistan
BE	Belgium	GN	Guinea	MK	The former Yugoslav Republic of Macedonia	TM	Turkmenistan
BF	Burkina Faso	GR	Greece			TR	Turkey
BG	Bulgaria	HU	Hungary	ML	Mali	TT	Trinidad and Tobago
BJ	Benin	IE	Ireland	MN	Mongolia	UA	Ukraine
BR	Brazil	IL	Israel	MR	Mauritania	UG	Uganda
BY	Belarus	IS	Iceland	MW	Malawi	US	United States of America
CA	Canada	IT	Italy	MX	Mexico	UZ	Uzbekistan
CF	Central African Republic	JP	Japan	NE	Niger	VN	Viet Nam
CG	Congo	KE	Kenya	NL	Netherlands	YU	Yugoslavia
CH	Switzerland	KG	Kyrgyzstan	NO	Norway	ZW	Zimbabwe
CI	Côte d'Ivoire	KP	Democratic People's Republic of Korea	NZ	New Zealand		
CM	Cameroon			PL	Poland		
CN	China	KR	Republic of Korea	PT	Portugal		
CU	Cuba	KZ	Kazakstan	RO	Romania		
CZ	Czech Republic	LC	Saint Lucia	RU	Russian Federation		
DE	Germany	LI	Liechtenstein	SD	Sudan		
DK	Denmark	LK	Sri Lanka	SE	Sweden		
EE	Estonia	LR	Liberia	SG	Singapore		

VIRTUALIZED MULTIMEDIA CONNECTION SYSTEM

Background of the Invention

The invention relates generally to multimedia connection systems.

With few exceptions, systems supporting facilities for the sharing of live audio and motion-video images have provided integrated hardware platforms that contained both video encoding/decoding devices and an Integrated Services Digital Network Basic Rate Interface (ISDN/BRI). Stand-alone, dedicated video conferencing systems began as wholly proprietary designs (early video conferencing systems from PictureTel Corporation and Compression Laboratories, Inc.), but by 1993, conferencing products based on the personal computer became available. These systems relied upon standardized operating systems to host command, control, and user interface software components. Specialized hardware support for videoconferencing services was implemented in adapter board configurations that contained real-time video processing facilities, integrated with an ISDN/BRI hardware component (PictureTel S1000, CLI Eclipse). For all intents and purposes, the software architectures used in the implementation of the motion-video connectivity sub-systems have been extremely tenuous, and essentially unusable as discreet modular components in that they lacked a comprehensive, extensible software model to support the diversity of underlying hardware technologies.

Perhaps the best attempt at creating a simple, reusable motion-video connectivity sub-system for integration into second-party products is available from Zydacron, Inc. (Zydacron Z240, Z250). Even the simple, clean implementation of this system's software control mechanism requires many months of specialized software

development, including significant design and implementation of complicated protocol components, before the sub-system is usable in a commercial product.

ITU-T RECOMMENDATION H.320

Motion-video connectivity, between systems from different vendors, is possible only through the general acceptance of *standardized protocols* for interconnection between local and remote stations. In March of 1993, the International Telecommunication Union (ITU-T) adopted Recommendation H.221 (LINE TRANSMISSION OF NON-TELEPHONE SIGNALS) as the standard for the interconnection of devices supporting the exchange of audio, video, and binary data types (see ITU-T Recommendation H.221, FRAME STRUCTURE FOR A 64 TO 1920 kbits/s CHANNEL IN AUDIOVISUAL TELESERVICES, 1994. This protocol is grouped with two other related interconnection protocols under the rubric of Recommendation H.320, which is now the generally accepted set of protocols for implementing composite audio/motion-video/data connectivity across the Integrated Services Digital Network as embodied in narrow-band visual telephone systems ISDN- see, ITU-T Recommendation H.320, NARROW-BAND VISUAL TELEPHONE SYSTEMS AND TERMINAL EQUIPMENT, 1994). Recommendation H.320 is a virtualized definition of an extensible, finite set of capabilities, device modes, data transfer frame structures, and call control procedures. At some level in the software layers that comprise H.320-compliant connectivity stations, there must be (by definition) an implementation of a significant subset of the Recommendation H.320 multimedia interconnection services. Despite this distinct commonality shared by inter-connectable stations, there are no commercially available software models, known to this author, that promote these standardized audiovisual/data connectivity

services to a useful, consistent, reusable kernel of device-independent software control elements.

SYSTEM ARCHITECTURES

System and user interface software designs for multimedia connectivity stations have typically been derived directly from the service profile of the underlying devices that they control -- multimedia connectivity software architectures are mostly hardware driven. Since multimedia connectivity tasks, such as videoconferencing, require synchronous encoding/decoding of audio and video data at high data rates emanating to/from synchronous data transfer connectivity devices, most of the motion-video connectivity devices integrate all the necessary components onto one large, multi-purpose device. Typically these devices take the form of an ISA or EISA personal computer adapter that includes additional hardware support for specialized video overlay functions. Without a major software development effort, it is impossible to utilize the manufacturer's sub-system for a new connectivity application. Those wishing to impart videoconferencing services to their enterprise are most frequently restricted to the single software application program provided by the hardware vendor; that is the only one capable of sufficiently driving the vendor's complicated hardware configurations. PictureTel Corporation, Zydacron, Inc., and Compression Labs, Inc., design and develop most of the world's motion-video connectivity sub-systems according to this multiple integrated device architecture. These systems perform well for stand-alone videoconferencing purposes. Recent attempts by Intel Corporation to produce software video solutions with minimal hardware support, are of inferior image and sound quality. The allocation of valuable central processing unit resources to real-time video

encoding/decoding tasks is inefficient at 128 kbit/s data transfer rates, and entirely inappropriate at transfer rates of 384 kbit/s, or more. A data transfer rate of 384 kbit/s is required to produce motion-video connection services with frame rates and image clarity sufficient for large-scale acceptance of these technologies; a 128 kbit/s data transfer rate produces video image quality that can best be described as "disappointing" to uninitiated technology customers who typically expect a broadcast quality television image (see D. Minoli and R. Keinath, Distributed Multimedia Through Broadband Communications Services, Norwood, Massachusetts, Artech House, pp. 187-207, 1994). There will likely continue to be a continuous stream of hardware and software solutions to improve the quality of motion-video connectivity using personal computers, and systems will continue to undergo rapid change as high-bandwidth ISDN connections become cost effective and generally available. In consideration of the extraordinary engineering effort required to produce motion-video connectivity systems, and the difficulties of developing software to support new devices, more can be done to reuse the high-level applications and protocol support code developed for these products.

Summary of the Invention

VIRTUALIZED SYSTEM DESIGNS

An analogous problem to that of the hardware-driven software designs in multimedia connection systems can be found in the early attempts to create Graphical User Interfaces for the wide variety of graphics hardware used with personal computers. Here again, hardware features influenced overlaying software designs, with a proprietary device driver interface supported by almost every distinct video graphics adapter manufacturer. The

Microsoft Windows Operating Environment, and the Operating System/2 Presentation Manager, solved the problems related to video graphics hardware variability by abstracting the services of these devices to a fully device-independent model. The original Microsoft solution is called the Graphics Device Interface (GDI), and it is the paradigm shift in video graphics technology made possible by this particular product component, that has helped promote the Graphical User Interface (GUI) to its ubiquitous market position. The invention of the GDI allowed the development of GUI applications that would run over any graphics hardware integrated according to the GDI's prescribed methodology (see C. Petzold, Programming Windows 3.1, Redmond, Washington, Microsoft Press, Chapter 11, 1992.

The principle that underlies the GDI is that there exists a finite set of graphical operations that will enable a software developer to draw just about anything on a bitmapped display device. It is taken into consideration that some graphics hardware is more or less suitable for the direct implementation of these operations; some operations may not be supported at all by the hardware. To derive a set of abstract, logical operations, without giving consideration to the underlying mechanism needed to support them, is referred to as the process of virtualization. In a GDI implementation, any graphics operation that may be supported directly by a hardware function, is accessed directly using a vendor-specific device driver. If a close mapping of a physical to a logical graphical operation exists, some software modification of the physical operation, to better implement the desired logical operation, is invoked as needed. If no hardware support exists for the operation, it may be emulated entirely in software, or marked as a task of which the

vendor-specific driver is incapable. A structured capability report mechanism is available to applications using GDI, so that they may determine if a specific operation is even possible. Regardless of whether a particular operation is supported by the graphics device per se, or can be emulated, if there is any way GDI can fulfill the request for service, then the graphical system is considered to be capable. This same virtualization principle behind the Graphics Device Interface can be expanded to create a logical description of operations necessary to fully describe multimedia interconnection operations.

A VIRTUALIZED MULTIMEDIA CONNECTIVITY SYSTEM

Videoconferencing is but one interesting multimedia connectivity service. However, what is needed is a change in the way we view multimedia interconnection, not only in terms of the logical operations we wish to perform, but in a way that most advantageously applies those operations to specific configurations of audio-video transducers, and diverse sources of synchronous data connectivity. A generalized model for multimedia connectivity application development must take into consideration that the essence of these technologies is the structured sharing of sound and light spectral data (as opposed to binary data). The vendor-specific facets of media transducers and network interfaces employed to implement related services must be rendered entirely irrelevant to the operation of software programs desirous of device-independent audiovisual teleservices.

In that regard, an efficient, consistent, and extensible presentation of multimedia connectivity services to software programs is achieved through the appropriate run-time binding of media transducers to

connectivity protocol stacks. Device-independent multimedia connectivity services are abstracted, in software, to an externally consistent media and network interface control interface. A single binary software object is constructed to encapsulate all relevant hardware and software components necessary to support virtualized multimedia connection services. These services are accessed through manipulation of the object's members. A specific instance of the virtualized, encapsulated media control and connectivity components required to implement the defined services, is referred to as a Virtual Connection Object (VCO). Each VCO contains a reusable Virtual Device Interface (VDI) software component that contains the VCO's Software Control Interface (SCI) and device-independent media/connection device control methods. The VDI derives implementation of its services from a Virtualization Layer (VL). The Physical Device Interface (PDI) provides control of the physical media transducers and one or more network interface units, in a fashion consistent with both the techniques specified by their manufacturers, and in a way that enables their efficient utilization by methods in the VL. Device-generated events, occurring in real-time, asynchronous with the system scheduler, are ~~inserted~~ into an infinite event queue, to be periodically dispatched to the VDI, synchronous with the system scheduler. Physical limitations on the level of service provided by encapsulated media transducers/network interface, are expressed as the Local Capabilities. When connected, the capabilities of the remote station are expressed as the Remote Capabilities, and are available to the constructor of the VCO. The quality of the connection is described as the logical intersection of both the Local Capabilities and the Remote Capabilities, and is referred to as the Connection Capabilities. VCO

implementations abstract multimedia connectivity services to the level of the Open Systems Interconnection Reference Model Presentation Layer; device-dependent control methodologies for vendor-specific media transducers and connectivity protocol stacks have no representation in the Presentation Layer. Software programs that construct VCOs and utilize the presented multimedia connectivity services, are referred to as VCO Clients. These device-independent connectivity programs realize the benefits of interoperability across any multimedia connectivity sub-system that encapsulates its services into a Virtual Connection Object, according to the disclosed methodology.

In general, in one aspect, the invention is a multimedia connectivity program residing in computer readable memory. The connectivity program when executed on a computer providing to an application program multimedia connectivity services through a real-time multimedia device control subsystem including components selected from among a plurality of multimedia devices and a plurality of real-time multimedia protocol stacks. The program includes a single binary object encapsulating a virtual device interface and a device control interface. The virtual device interface includes a plurality of virtual methods that represent logical operations available to the application program for controlling the multimedia device control subsystem. The plurality of virtual functions are completely independent of the components within the device control subsystem. The device control interface mapps the plurality of virtual functions to physical control methods which control the components of the multimedia control subsystem.

In preferred embodiments, the device control interface includes a plurality of media control objects which represent audiovisual and binary data streams

associated with the components of the plurality of devices and/or real-time multimedia protocol stacks. The virtual device interface is configured to present a logical representation of the multimedia connectivity services provided by the connectivity program. The device control interface includes a virtualization layer and a physical device interface layer, the virtualization layer being located between the virtual device interface and the physical device interface. The physical device interface directly interfaces to the device control subsystem to provide a physical implementation of services requested by the application through the virtual device interface. The virtualization layer resides between the virtual device interface and the physical device interface layer and is configured to translate and map device control mechanisms employed by the underlying multimedia control sub-system to representations required by the virtual methods of the virtual device interface.

Also, in preferred embodiments, the plurality of media control objects provides the multimedia connectivity control program with a pool of media device signal resources. Each of the plurality of media control objects is classified as at least one of type of the group consisting of an audio type, a video type, an image type~~, and~~ a binary data type. Also, each of the plurality of media control objects represents a signal from the group consisting of a signal from a remote station, a signal to a remote station, a signal from a local output device, and a signal to a local output device. The operations performed on the plurality of media control objects by the physical device layer under control of the virtual device interface implements a logical software switching mechanism. The virtual device interface implements a plurality of public member functions, the virtual functions being a subset of those

public member functions and the plurality of public member functions representing all of the public member functions in the single binary object that are accessible by the application program.

In general, in anotehr aspect, the invention is a computer programmed to provide to an application program multimedia connectivity services through a real-time multimedia device control subsystem. The multimedia device control subsystem includescomponents selected from among a plurality of multimedia devices and a plurality of real-time multimedia protocol stacks. The programmed computer includes a virtual device interface and a device control interface, both of which are encapsulated in a single binary object. The virtual device interface includes a plurality of virtual methods that represent logical operations available to the application program for controlling the multimedia device control subsystem. The plurality of virtual functions are completely independent of the components within the device control subsystem. The device control interface maps the plurality of virtual functions to physical control methods which control the components of the multimedia control subsystem.

In general, in yet another aspect, the invention is a computer implemented method of providing multimedia connectivity services through a real-time multimedia device control subsystem. The multimedia device control subsystem includes components selected from among a plurality of multimedia devices and a plurality of real-time multimedia protocol stacks. The method includes the steps of defining and supporting by computer implemented steps a virtual device interface; and defining and supporting by computer implemented steps a device control interface. Both the virtual device interface and the device control interface are encapsulated in a single

binary object. The virtual device interface includes a plurality of virtual methods that represent logical operations available to the application program for controlling the multimedia device control subsystem. The plurality of virtual functions are completely independent of the components within the device control subsystem. The device control interface maps the plurality of virtual functions to physical control methods which control the components of the multimedia control subsystem.

Multimedia connectivity sub-systems, when developed for use in a VMCS, present an externally consistent, fully abstracted set of logical operations to software programs, therewith exposing the faculty to create adjunct, device-independent, interoperable multimedia connectivity software application programs. The disclosed invention is a methodology to enable the structured sharing of spectral information between interconnected microcomputer stations. Though principally intended to facilitate control of live audio and (motion) video information, this comprehensive connectivity paradigm incorporates mechanisms for store-forward operations; binary data object transfers are also supported. For the purposes of practical consideration, the VMCS pursues a classic client-server model of application/sub-system interaction. The sub-system presents a coherent software control mechanism to device-independent connectivity applications created explicitly to utilize its structured, highly-typed, set of services.

Insofar as software programs benefit from virtualized binary data sharing technologies, the same benefits may be realized by those sharing spectral information, if the system is implemented as a Virtualized Multimedia Connection System (VMCS).

Other advantages and features will become apparent from the following description of the preferred embodiment and from the claims.

Brief Description of the Drawing

Fig. 1 shows the symbol conventions used in the following figures;

Fig. 2 is a block diagram showing a VCMS component overview;

Fig. 3 is a block diagram showing a VCO architecture overview;

Fig. 4 is a block diagram showing the VCO software architecture;

Fig. 5 is a block diagram showing the VCO client application software architecture;

Fig. 6 is a block diagram showing the VCO class derivation;

Fig. 6A is a table of the VCO exception handling modalities;

Fig. 6B is a table of the VCO trace output modalities;

Fig. 6C is a table of VCO events which trigger notification;

Fig. 7 is a block diagram showing the device control mechanism;

Fig. 8 is a block diagram showing the VCO control methodologies;

Fig. 9 is a block diagram showing the terminal output control flow;

Fig. 10 is a block diagram showing the signal triggering mechanism control flow;

Fig. 11 is a block diagram showing the event queuing and dispatching control flow;

Fig. 12 is a block diagram showing the connection capability control flow;

Fig. 13 is a block diagram showing the capability and mode mapping control flow;

Fig. 14 is a block diagram showing the call controller control flow;

Fig. 15 is a block diagram showing the line disconnection control flow;

Fig. 16 is a block diagram showing the line dialed, ringback, busy, and connected control flows;

Fig. 17 is a block diagram showing the line ring control flow;

Fig. 18 is a block diagram showing the call reset, mode setting, and mode restoring control flows;

Fig. 19 is a block diagram showing the exception handling control flow;

Fig. 20 is a block diagram showing the media control object command control flow;

Fig. 21 is a block diagram showing the media device capability query control flow;

Fig. 22 is a block diagram showing the media access control request control flow;

Fig. 23 is a block diagram showing the device event processing control flow;

Fig. 24 is a block diagram showing the object construction and destruction control flows;

Fig. 25 is a block diagram showing the "Open" command control flow;

Fig. 26 is a block diagram showing the "Close" command control flow;

Fig. 27 is a block diagram showing the "Call" command control flow;

Fig. 28 is a block diagram showing the "Multicall" command control flow;

Fig. 29 is a block diagram showing the "Hang-Up" command control flow; and

Fig. 30 is a table of the multipoint control operations.

Appendix: contianing source code for Virtual Device Interface Header File,

Description of the Preferred Embodiments

DEFINITIONS

Provided below are definitions for terms used throughout this disclosure. While common technology parlance may assign a variety of alternative meanings to them, those definitions following refer to specific usages in this manuscript only, noted explicitly to alleviate ambiguous technology references henceforward.

Transducer

This term refers to a device that converts one form of energy into another. Here specific reference is given to those that convert light and sound energy to electrical pulses, or inversely, electrical pulses back to light and sound energy. Examples include cameras, microphones, speakers, and video monitors.

Signal

This term refers to a digital data stream used to transfer raw or encoded binary information, except in the case of direct references to "bit-rate allocation signal indications."

Indication

This term refers to a message connoting a change in state or modality of a system or station component; basic unit of notification used for the sole purpose of communicating the occurrence of some specific event.

Notification

This term refers to an indication transmitted from one software component in the system to another software component in the same system. Typically used to notify software system components that some specific event has occurred and some response is required.

Spectral information
This term refers to sound and light data that are represented as modulations of electromagnetic or audible spectra; audible and visible waveform information.

Binary information
This term refers to electrical pulse data encoded as binary numerical values that are typically referenced in octets.

Terminal
This term refers to as a physical or virtual teletype console device that displays text data output to it, and returns text input, such as if read from a keyboard device; essentially a dedicated text-processing I/O device or software representation thereof, with no significant processing ability.

Station
This term refers to an intelligent node on a network to which other network nodes can connect and exchange messages.

Local station
This term refers to the station whose resources are directly addressable without using an intervening network connection; conceptually perceived as the near-end of any connection.

Remote station
This term refers to the station whose resources are not directly addressable without an intervening network connection; conceptually perceived as the far-end of any connection.

Sharing
This term refers to bi-directional data transfer between interconnected stations on a network.

Vendor-specific
This term refers to any hardware or software system component that requires a non-standardized software control layer to accommodate the particular requisite interface format and control procedures described by its manufacturer.

Multimedia
This term refers to a class of digital computer technologies that store, retrieve, manipulate, and play back audible and visual information. These technologies are embodied in combined software and digital hardware sub-systems that encode spectral information presented to input transducers, into digital data streams that are then stored in a compressed format. This compressed digital information is later reconstituted back to spectral information by decompressing, decoding, and subsequent retransmission through output transducers.

Connectivity
This term refers to the generalized concept of establishing a communications link between two or more stations on a network, exchanging messages according to some preconceived notion of structured interaction; this notion of interaction referred to as a protocol.

Multimedia connectivity

This term refers to the generalized concept of establishing an audible, and/or visual communications link between two or more stations on a network; These technologies are embodied in combined software and digital hardware sub-systems that encode spectral information presented to input transducers, into digital data streams that are then transmitted across a network connection to a remote station in a compressed format. There it is reconstituted back to spectral information by decompressing, decoding, and subsequent retransmission through output transducers. When occurring bi-directionally in real-time, without delay, the connection between the stations is described as "live", as if the input transducers of one station were connected directly to the output transducers of the other, and the network becomes conceptually irrelevant to the process. A variety of other media forms, such as still pictures and plain binary data, may also be exchanged between stations on an occasional basis, and played back as needed.

Media Control Interface (MCI)

This term refers to a device driver interface specification that allows its users to control underlying physical media devices using a somewhat well-defined, standardized string-token command language.

Media Access Control (MAC)

This term refers to that set of MCI drivers that provide a standardized physical device control interface layer between the more device-independent software layers that issue MCI string commands, and the vendor-specific device drivers that contain proprietary interfaces and control procedures to initialize, shutdown, and utilize the peripheral hardware components.

Object-Oriented Design

This term refers to a methodology to enhance the quality of computer systems by describing their constituent components as discreet sets of related, well-defined operations whose implementations are isolated from their functionally described operational profiles; interactions between system software components are defined with equal precision, according to a specialized software development methodology. Highly-structured programming languages1 and design tools promote creation of modular, reusable software components that can be recombined to build new components; existing components are better understood in terms of functionality and reliability. In this way, implementations of new components borrow the functionality (and demonstrated reliability) of preexisting software technologies to create new products, thereby significantly reducing development time and dramatically improving overall system reliability.

OPERATING SYSTEM MODEL

A more accurate technical description of the VMCS is that of a Virtualized Multimedia Connectivity Operating System. Designed for a multithreaded, protected memory model implementation, the VMCS server component runs parallel to the client applications that utilize it, the server responding to client requests with a stream of notifications directed to class objects located in the client programs. A client application invoking a VMCS server, spawns a concurrent operating system that effectively manages all hardware and software components necessary to establishing a device-independent multimedia connectivity service, in much the same way as any operating system does to support its general application programs. Once created, the VMCS server runs by itself, independent of the client, and capable of offering

services to more than one client at the same time. Just as with advanced operating systems, the transactions between clients and servers are fully protected, and highly structured with regards to both syntax and sequence. The VCO Client selects an "operating system" that best suits the system hardware configuration, invokes it when needed, discards it when it is no longer needed, and changes it when it prefers an "operating system" with a different service profile.

Consistent with the intention of its design, multiple VMCSs can be concurrently invoked. The VMCS, in accordance with the "operating system" model heretofore described, was intended from its inception, for implementation in multiprocessor or distributed systems where a separate coprocessor, parallel processor, or entire system could house the server component separately. Further, embedded implementations (for use with coprocessor systems) do not pose the usual implementation difficulties associated with sub-system designs whose client and server component were assumed to reside in shared memory.

STANDARDS COMPLIANCE

There is a well-defined modality of interaction between VMCS servers, and the VMCS applications that use them, the orders of whose operations are specified with mathematical precision. Resultantly, there is a high-degree of predictability in the progression of connectivity sessions, and corresponding measurable improvement in their robustness. VMCS implementations are unique in the domain of multimedia connectivity systems. Specifically, they make possible the creation of standardized software communication products, enable connectivity applications to run over any compliant connectivity sub-system, fully integrate audiovisual/data

connectivity services into a single control mechanism, reduce software development time, and expose extraordinary levels of derived functionality.

This methodology is primarily a software development technique. Object-oriented software components are created to abstract the low-level services of media control and network interface components to a fully-virtualized multimedia connection service that integrates the ITU-T multimedia interconnection protocols (H.320), the Open System Interconnection (OSI) Reference Model, and object-oriented software design principles. The VMCS architecture combines these paradigms into a dynamically instantiable client-server multimedia connectivity service technology. ITU-T Recommendation H.320 defines a discreet set of operations and procedures necessary to the sharing of spectral and binary data between compliant interconnected stations. It enables reliable, structured data transfer and device mode synchronization to stations connected to the Integrated Services Digital Network (ISDN). A VMCS implementation employs the ubiquitous H.320 recommendation as a rigorous definition of its most basic set of multimedia connectivity operations. For stations that access the ISDN, this application of H.320 is natural and obvious, but the VMCS goes on to take further advantage of the apposite H.320 structure. The VMCS architecture insists upon the promotion of the H.320 protocol set to that of a universal framework to which even non-compliant protocols can be promoted.

ARCHITECTURE

OSI REFERENCE MODEL

Connectivity system developers can abstract the presentation of non-compliant services to a representation more efficiently managed by applications

(that are typically unconcerned with the specific requirements of the control mechanisms). The Open System Interconnection (OSI) Reference Model defines a layering model offered internationally as a generalized system architecture to affect the designs of connectivity software systems, particularly with respect to host stations desirous of reliable interconnection. For any host operating system (OS), a Virtualized Multimedia Connection System (VMCS) provides an exact definition of the OSI Presentation Layer that serves as the interface between a connectivity application (client), and the connectivity sub-system (server). The majority of software components, common to all VMCS implementations, are reusable versions of the services specified to reside in the OSI Session Layer; they provide device-independent implementations of the protocols represented by the H.320 rubric. Media control and network interface manufacturers are usually best qualified to supply low-level device/protocol support drivers that comprise the OSI Transport, Network, and Data Link Layers. The VMCS is most efficiently implemented when it leaves this task to them, and instead builds on their work. For specific media control and connectivity tasks, it is often indeterminate as to whether a device, or software component, will comprise the best utensil. Since the OSI Reference Model is functionally specified, a system developer has the flexibility to derive a VMCS sub-component, or entire OSI Layer, in a way that best supports its design specification, regardless of whether it is build with existing or newly created sub-components.

OBJECT-ORIENTATION

Both VCO Client and server software components are developed with an object-oriented programming language,

according to a precisely-defined class inheritance and derivation hierarchy. A binary software object is created to encapsulate disparate software components into one functional entity, whose services are available only through structured access of its control elements (member functions and member data). The strategy of all VMCS designs, derived by the application of this methodology, is to encapsulate media control services, connectivity services, and protocol support components, together, in a way that best integrates their properties into an instance of a standardized multimedia connection service to be used by device-independent client applications. Specific protocol support procedures, and media control components, are shared by all VMCS implementations; usually they are worth preserving, fine-tuning, and carrying forward to new implementations. VMCS implementations created to support new hardware configurations are most accommodating to this circumstance, to the extent that they are typically minor modifications of a reference VMCS implementation. New VMCS implementations may be designed in one of three modalities. First, a VMCS can be developed to exploit the services of an existing multimedia connectivity product (hardware and software sub-system layers) manufactured by a third party. Such a project would restructure proprietary controls of the native interface, coercing (virtualizing) them to the structured consistency defined by the VMCS paradigm. A second modality is to create a presentation of the defined VMCS services for a new device set, creating all device control components, VMCS components, and perhaps the devices themselves. Lastly, designs can, at run-time, invoke a particular presentation of services, derived from the ad hoc association of media control devices with selected

connection services, in a way most suitable to producing a desired multimedia connectivity service profile.

CLIENT/SERVER MODEL

Notwithstanding the flexibility afforded by software implementations, it is useful to describe the works of the VMCS in terms of discreet, mechanical components. There is no requirement for any component to be implemented entirely in hardware, or entirely in software, per se, so the distinction will not be brought to bear on this discussion. There are two major components in any VMCS: the multimedia connection sub-system (server), and an application program that invokes its services (client). Any VCO Client application can invoke the services of any server, without hesitation, with a guarantee of compatible access. More nebulous is the parity between the quality of services requested by the client, and those provisioned by physical devices encapsulated in the server. Strictly speaking, the server is the binary software object that encapsulates the media control and connectivity sub-system. It is referred to as a Virtual Connection Object (VCO), and the client application that invokes usage of its services, is referred to as a Virtual Connection Object Client Application (VCO Client). In this section, the scope of the discussion will be limited to a functional description of these entities.

SERVER COMPONENTS

Virtual Connection Objects are binary software objects that encapsulate the media control and connectivity components requisite to the implementation of a discreet subset of multimedia connectivity services. They are invoked and adjusted by manipulation of their members. When constructed, a VCO dynamically builds the

particular operational context of hardware and software components needed to best implement virtualized multimedia connection services. Destruction of the object deletes this operational context by shutting down all components, turning off devices, and releasing any allocated resources. For the host OS, every implementation of such an object presents members that are identical in syntax, structure, and usage. In fact, every VCO is functionally analogous in its control mechanism, but unique in both its name, and the degree to which it is able to realize the full set of VMCS services defined to be represented in every instance thereof; that is each VCO has a unique set of capabilities. Inherent to every VCO, is the ability to provide metrics describing the potential quality of services locally available, and the actual quality of services available while connected to a particular remote station. The service profile of the unconnected local station is available prior to device initialization (but after VCO construction), for casual examination by interested VCO Clients; this profile is referred to as the Local Capabilities. The service profile of a remote station is available while it is connected to the local station, and this profile is referred to as the Remote Capabilities. Also available when connected, is the service profile of the connection itself, referred to as the Connection Capabilities. This is the preeminently useful metric, and quantifies the potential quality of interactions between the local and remote stations; precisely, it is the logical intersection of the local and remote capabilities.

A real-time reporting mechanism alerts system software objects of changes in the specific states, modalities, and conditions critical to the operation of the encapsulated multimedia connectivity sub-system. The basic unit of information, or indication, is referred to

as an Event, and each VCO contains a specialized delivery system that can notify software components in the host system when one such event has occurred; a Notifier Object is the mechanism for this delivery. Notifier Objects are triggered by the occurrence of any event type to which they are programmed to respond, and they are used both internally by the VCO, and externally by its clients. Finally, it should be pointed out that a VCO is implemented to present the services of one particular configuration of media control/network interface setup that comprises the multimedia connectivity sub-system, and it is likely each significantly different hardware and/or software configuration will require a somewhat different VCO implementation; a VCO is a device-dependent entity.

CLIENT COMPONENTS

Applications programs described in the VMCS are referred to as Virtual Connection Object Clients, are device-independent, software application programs that invoke VCOs, and manipulate their members to control live information-sharing sessions with remote stations; to that end, they create use-specific logical combinations of currently available audiovisual/data resources to support a defined set of multimedia connectivity tasks that is the embodiment of that particular connectivity application. They are developed in an apriori fashion, according to a concise, multimedia connectivity software control interface specification, the integral implementation of which each VCO must contain. Clients dynamically invoke at least one appropriate VCO, selected (from a library) according to some notion of suitability, and then construct it only when a connectivity session is actually to be initiated. VCO Clients create instances of Notifier Objects, and utilize them as a mechanism to

respond (more-or-less instantaneously) to occurrence of divers events to which they have been rendered sensitive. A client software object that contains member functions to receive, and respond appropriately to, these signaled events, is referred to as a Notification Receiver Object. Clients may monitor and/or intercept connectivity and device control related occurrences in the encapsulated sub-system, by creating instances of VCO Notifier Objects with specific response profiles. These Notifier Object response profiles may be reconfigured ad hoc; they define the set of specific events that will trigger notifications (when a specified event occurs) to one identified NRO. There can be only one NRO associated with a particular instance of a Notifier Object. In a given host OS, any VCO Client can function using any VCO; a VCO Client is a device-independent entity.

METHOD

Here follows description of a method to implement a preferred embodiment of a Virtualized Multimedia Connection System (VMCS). The scope for the disclosure will be restricted to an elucidation of those elements required to derive a design specification for a fully device-independent multimedia connection system; a system to be built from third-party media control devices, device drivers, and a connectivity protocol stack running over a network interface unit. All VMCS software components are created with an object-oriented programming language (see M.A. Ellis and B. Stroustrup, The Annotated C++ Reference Manual, Reading, Massachusetts, Addison-Wesley, 1990). Attendant source code provides a precise definition of the host/client software interface, and an example of a simple, portable device-independent connectivity application.

The design of this system will be considered initially from the perspective of an overview, and subsequently as a functional examination of its component interworkings. Next comes a detailed examination of critical software and hardware constructs needed to physically implement a VMCS design. The topic concludes with a discussion related to the deployment of a working system in an actual host computer system. For this section in particular, the examinations of system details remain more generalized in the start, and are more fully later resolved, the strategic intention to permit a system engineer to pursue a top-down design approach for his particular VMCS adaptation. Compliant designs will produce products supporting exact binary compatibility between every VMCS created for the same target operating system, and exact functional compatibility between every VMCS, regardless of the target operating system.

DESIGN

VISUAL TELEPHONE SYSTEM MODEL

The creation of a visual telephone system is the most likely application for a VMCS implementation. Such a system illustrates all of the basic components that comprise the set of media control, network interface, and software control features common to most such solutions now available. The ITU-T describes a generalized model for this type of system in a document referred to as Recommendation H.320. This discussion, as related to a VMCS implementation, is best served by a similar treatment of the subject, as it relates to the task of deriving a virtualized model of multimedia connectivity; the VMCS software abstraction represents the functional aspects of the generalized visual telephone, with some notable extensions to its base level utility. It must also be pointed out that a VMCS implementation in no way

relies upon the ITU-T recommendations below the level of the signal and indication protocols specified by H.320 -- any network or connectivity sub-system could conceivably be adapted as the underlying network transport layer.

ELEMENTS OF A VISUAL TELEPHONE SYSTEM

The prototypical VMCS system relies on divers hardware and software components to transfer audio, motion-video, still pictures (images), and binary data objects between stations connected to the same network. Devices and control systems described below should be considered in terms of being functional entities; the potential to support device characteristics with a software emulation module is an implementation alternative that does not alter the basic strategy of VMCS design. In fact, input from, or output to a transducer device can be simulated by a data stream read from, or written to backing store.

I/O Equipment

These devices are referred to as transducers in that it is their primary task to convert energy forms from spectral to pulsed electrical, or vise-versa. In practical terms, these devices are the only parts of the system with which the user interacts directly.

- **Audio devices** include stationary microphones and related sound detection equipment to input the voice of the user. For output, loudspeakers and headphones are output transducers in a VMCS.
- **Video devices** include a camera to input motion-video images of the user. For output, video monitors display motion-video, as do dedicated analog (NTSC/PAL) video displays

- **Image devices** include a camera to input images, as well as scanners to capture high-resolution images. For output, video monitors display images, and in addition, printers that can render images are considered output transducers.
- **Data devices** do not convert energy forms, but are essentially "locations" in the system through which data flows, or in which it is stored. Data devices include any backing store (disk) entities, or data ports, such as a communications port. Dedicated data streams, or "socket" interfaces would also qualify as valid VMCS inputs/outputs for binary data.

CODEC

These devices compress data from input transducer devices prior to transmission or storage, or inversely decompress data following transmission from a remote station or retrieval from storage. The process of compression is referred to as encoding, as spectral data is encoded according to an algorithm; decompression is correspondingly referred to decoding. Visual telephones transmit and receive audio and video data in a compressed format so as to enhance the efficiency of the system in its endeavors to exchange large volumes of audio and video data across connections that only support a very limited bandwidth. Storage of images to disk proceeds in the same way to reduce the amount of storage space required.

- **Audio de/compression** devices are typically incorporated together with H.221 encoding/decoding hardware used for video processing; audio compression and

decompression are often accomplished in software.

- **Video de/compression** for live motion-video conferencing is best accomplished with hardware specifically designed for H.221 encoding/decoding; video compression in software (for high quality video) is typically unsatisfactory.
- **Image de/compression** for images is typically done in software according to a specialized algorithm (such as JPEG) or transmitted as a simple bitmap.
- **Data de/compression** is typically not required, save only when the data object itself has been compressed prior to transmission, as part of a separate operation.

Telematic Equipment

These are facilities (devices or software components) that act a visual aids to enhance conferencing. Application sharing features, common whiteboards, and image transceivers are included in this category, as are scanners and facsimile devices attached to communications ports on the visual telephone station.

Multiplexors/Demultiplexors

Signal multiplexing and demultiplexing operations are typically combined into a single hardware device that combines the audio, video, and related data streams to the single H.221 frame structure (multimedia signal) used in line transmission, and restores them to their constituent data streams upon receipt from a remote station or storage entity.

Network Interface

This component is typically a hardware device that provides the physical connection between the host station and the network, though software simulations may provide an emulated version of the same.

Network

Any transport medium supporting actual or emulated line transmission of the H.221 signal and indication portion of this protocol, and capable of synchronous (and simultaneous) audio, video, and binary data transport. Examples of recommended network types include the following:

- ISDN (Basic and Primary Rate Interfaces)
- B-ISDN (Using ATM)
- LAN (FDDI and high-bandwidth proprietary technologies)
- Mobile/Radio (and other wireless)
- Analog (PSTN)

Multipoint Control Unit

This is a specialized device whose functionality is described by ITU-T Recommendations H.243 and H.231. The functionality of this device must be available for an implementation of a VMCS that is fully capable of the entire set of VMCS-defined multipoint control operations.

System Control

The VMCS is an implementation of this (system control) visual telephone system component, but conceived as a more generalized model suitable for utilization in a broad range of concurrent audiovisual/data sharing applications. The VCO component of the VMCS allows a device-independent connectivity client application to

utilize any audiovisual device control system services related to those of the defined visual telephone system.

VISUAL CALL PROGRESSION

A VMCS session takes the form of a visual telephone call. This interconnection procedure is precisely defined, and permits interoperability between dissimilar stations sporting diverse equipment complements, while retaining compliance to ITU-T Recommendation H.320. It would not be possible to utilize the plethora of potential operating modalities of VMCS systems without a thorough categorization of the set of all those modalities known to be supported by the local station's equipment configuration. Further, each station participating in a connectivity session (call) must come to an understanding of the modalities supported by its counterpart at the other end of the connection. What ensues at the time of the call is a planned negotiation that pits the performance expectations of the local station (as to the set of operating modalities it would ideally prefer to assume during the session), against the cataloged limitations of its remote station peer; a common set of operating modalities they both must co-operatively determine and ultimately share for effective audiovisual communication.

Establishment of Visual Telephone Call According to H.320

Shown below are the basic steps common to establishing a "normal" visual telephone call. This sequence is idealized in that it does not suffer interruption or complication as frequently occurs in actual use. Notwithstanding the inevitable anomalies encountered during live operation, the following sequence provides a model for call progress; one that must be implemented by every VMCS connectivity sub-system, and

one derived directly from the ITU-T visual telephone call procedure as follows:

1) **Call setup, out-band signaling (H.200)**

A request is made to the network for a connection to a remote station. Indication is received at both ends to indicate when this has occurred, and a bi-directional data channel from end to end is opened for use. This data channel carries multiplexed multimedia data between the stations. A device-independent call control mechanism is integral to all VMCS server components (Virtual Connection Objects) and operates directly to manage the call states and related operations required to create the connection and data channel.

2) **Mode initialization on initial channel (H.242)**

An exchange of device capabilities takes place once the connection is created and frame alignment for the data channel is achieved. It is at this time that the VCO call controller initiates sending its own capabilities to the remote station. It also receives and stores capabilities sent from the remote station. A common base-level audio mode is selected by the stations according to availability indicated by the exchanged capability sets. In the VMCS, any connection established must engage in the exchange (or emulate the exchange) of a capability set. A minimal bi-directional data channel must also be established in this phase. A capability exchange between stations can be initiated from either end, and may reoccur at any time (while connected) to assert a new ability recently assumed by a

station; a device may be turned on or off during the session, thus changes the stations capability profile.

3) Call setup of additional channels (if used)

Though not always used, the VCO supports connections of two separate lines. While network configurations may use six or more separate lines for a call, they are typically bonded together as one call and thus appear as a single line call to system call control. Call setup for additional lines proceeds in an identical fashion to that for the initial channel.

4) Initialization on additional channels (if used)

Though only used if additional channels have been established, capability exchange and mode selection proceeds as for the initial channel.

5) Establish common parameters (H.242)

Both stations have available a list of the other's capabilities. Each VMCS has associated with it, a specific conference profile parameter that is used to specify the operators preferred operating properties. Examples include a bias towards better video quality, or better audio quality. Coupled with this profile is a set of operating modalities that can be supported by both ends of the connection, and the preference of the remote operator. According to the physical limitations of the remote station's device capabilities, and in hopeful compliance with the preferences of the operator, a device mode negotiation mechanism seeks to algorithmically

deduce the most suitable set of device modes for the call by interacting with the remote station to realize them.

6) Visual telephone communication (H.261)

This phase refers to the fully established session itself, whereby audio, video, and binary data exchange can take place over existing data channels. The H.320 Recommendation states that visual and audio communications must be active in this phase, though the VMCS allows for idle connections during which time no data exchange takes place; that is audio and video may be disabled, while the connection is still technically active.

7) Termination

When one user hangs up, an indication is sent to the other end to signal termination of the call, in the form of disconnection signals for all lines. Such action by one end, initiates the call termination process; the initiating station shuts down all of its data transmission to the remote station. Out-band signaling is typically used for the purpose of disconnection.

8) Call release

Once termination has been initiated (when the other end receives this message) the station sets data transmission to idle for each disconnected channel. When all lines in the call have received disconnection events and been set to idle, the call is considered to be disconnected. Individual lines can be added or removed as needed, without disconnecting the entire call.

VMCS SYSTEM OVERVIEW

FIG. 2 depicts an overview of the major components of a Virtualized Multimedia Connection System. There are four significant entities shown: The Virtual Connection Object (VCO), the Virtual Connection Object Client Application (VCO Client), I/O devices, and the network. The VCO and the VCO Client are the subject of this disclosure. The I/O devices are connected with a direct physical link to adapter boards in the host system that permit the physical control of the I/O devices via device driver. Essentially, the only link the VCO has to these devices is through a vendor-specific Media Access Control software layer (or some other device driver), and the VCO link to the network interface unit is through a standardized, or vendor-specific network protocol stack. The network protocol stack shown in the drawing is likely some OSI Transport Layer abstraction of a connection service.

Summary of VMCS System Overview

From a conceptual standpoint, the VCO combines the services of the data transfer (network) service with associated media transducer devices, so as to reliably offer system command and control of a derived multimedia connectivity service to an application program. There are two primary components of any VMCS; they are as follows:

- **The Virtual Connection Object (VCO)** encapsulates all of the server components that are required to abstract a virtualized representation of the media control and connectivity sub-systems, offering it to device-independent connectivity clients.
- **The Client Application (VCO Client)** constructs and utilizes the virtualized multimedia connection services hidden away

inside the VCO by manipulating its member functions and member data. There are a number of specific objectives motivating this object-oriented system design, the substance of which receives commentary below:

System Design Objectives

Design-level Support for VMCS Services

The objective of the VMCS design is to provide design-level support for a full virtualization of any multimedia connection system, so that device-independent representations of a logical control mechanism could be ported to a wide variety of supporting media control devices and network interface. Client applications designed to utilize VMCS technology are interoperable with every VCO implementation (running under the same operating system) and thus more efficiently distributed, maintained, supported, and redeployed with new device complements. Most important is the ability to bind any network interfaces to any media control interface by the addition of specialized hardware and software layers that combine the functions of these components without affecting its mechanism of control. New and different underlying technologies are well utilized, however the extensible VMCS virtualization paradigm leaves their often peculiar operating procedures fully obscured from the view of application programs.

Progressive Isolation of Sub-system from Applications

The VCO component of the VMCS incorporates a number of design methodologies whose purpose is to dissociate the control of services, from their physical implementations. The Open System Interconnection (OSI) Reference Model (see B. Hebrawi, OSI Upper Layer Standards and Practices, New York, NY, McGraw-Hill,

pp.11-15, 1993) and object-oriented programming languages are primary building blocks in any VMCS.

- **OSI Layering** underpins the structure of the VMCS in that the VCO is a logical encapsulation of all of the OSI layers below the presentation layer. The VCO members themselves comprise the OSI Presentation Layer implementation, whereas device-independent routines in the upper VCO layers form a discreet and reusable OSI session layer.
- **Class Structure** of the VCO is rigidly defined so as to preserve the largest number of functional source components from modification, and to maintain the design integrity of the VMCS. Class components allow for a definitive specification of distinct, isolated functional units that will not affect their interactions with related components when their internal implementations have been modified to accommodate the operational characteristics of vendor-specific components.
- **Discreet Member Profile** preserves the modality of the interface that makes each and every VCO implementation appear the same to clients, and consequently it is most essential to maintain its form to enable interoperability for all clients. VCO Clients assume that the specific member profile of every VCO is identical, thus each can be invoked and utilized without concern for incompatibilities between them.
- **Token-based Job Description Language** is a text-token representation of the VCO member

functions and their highly structured enumerated arguments. If the structure of the VCO interface is consistent (over time) and reducible to simple string commands, then it is possible to reduce the control of any VCO to a series of text messages in a character stream. From this approach is derived the capacity to remotely control a VCO. User applications are almost entirely isolated from the server component in that each server (VCO) functions as a command interpreter.

Client Components

The VCO client in FIG. 2 is a device-independent layer that is dynamically portable at run-time to other VCO-encapsulated multimedia connectivity sub-systems. All VCO Clients are fully interoperable with all server (VCO) layer components that offer a consistent presentation (OSI Presentation Layer) constructed according to an interface specified in this document. Notification calls from the VCO to the client can occur spontaneously, asynchronous with other system events, and concurrent with notifications emanating from other VCOs invoked by the same client, thus most client modules should be designed to support re-entrancy and mutual-exclusive access to resources. A multithreaded implementation strategy is most efficient to manage the various, often concurrent tasks related to simultaneously maintaining the visual information presented to the user, and supporting the command, control, and real-time monitoring of the multimedia connectivity sub-system. Regardless of the frequency of device interrupt requests or the rate of message passing between interconnected stations, the flow of notifications from the VCO to the client is conducted according to a dynamically configurable interval that a

client can optimize according to its particular needs. In reality, the client runs at operating system speeds determined by the system scheduler, while low-level device control components in the VCO run at device speeds determined by the network and the devices themselves. Resultantly, VMCS systems share the following characteristics:

- Applications can vary the periodic rate at which they are actually notified of device events occurring sporadically, in real-time
- Applications never miss events, and are never unable to respond to them due to their occurring in time slices not allocated to the application. The VCO notifies its client application that a particular event has occurred, by scheduling the application to run, in response to that event, on a special thread created by the VCO event dispatcher.
- Application processing of device events catches up to the rate at which the device produces them, by continuing to send notification of events to the application during I/O latency periods when the device is less active.
- Critical section handling, as related to device interrupts and the management of device driver queues, is fully managed by the VCO, thus the application may process notification events at a high-level; it is, by design, nearly impossible for an application to corrupt the timing and real-time operation of the low-level device control sub-system. The application sees these sub-systems as a stream of interesting

events, none of which requires attention for proper VCO operation.

Application Shell

The application shell is preferably an event-driven graphical user interface (GUI) program written in an object-oriented programming language. There are no special considerations for VMCS integration. Retrofitting of existing GUI programs, to work as VCO Clients, is easily accomplished. In fact, any application shell that constructs a VCO is defined as a VCO client, and only a modicum of member functions need be called to establish a fully operational connectivity session to a remote station. A daemon that constructs a VCO, and fields string commands from a hardware or software data port, can serve as a non-interactive VCO client.

Notification Receiver Objects (NRO)

These software objects are created to provide a structured mechanism for responding to VCO events. Any application class object can contain a specific member function and adjunct function mapping component to direct the VCO to send a notification message to that object upon the occurrence of any set of specified events.

VCO Components

The VCO utilizes a three-layer software design that converts the native modalities and properties of some set of physical devices to those that can be accurately described using the parameters specified by the VCO. Underlying the software components are device control components used to physically operate the media control and connectivity sub-systems. These low-level components, and the devices that they control, are

typically provided by a third party specializing in their respective technologies.

Virtual Device Interface (VDI)

Provides a logically-defined, or virtualized interface to services that would be offered by an idealized, functionally advanced combination visual telephone/imaging system. The VDI is a reusable shell that is common to all VCO implementations (in a given OS).

Virtualization Layer (VL)

Translates logical operations defined in the VDI, to physical implementations (of those most similar) provided by the PDI. The VL is set of mostly reusable routines that are, as needed, partially reimplemented to work with a particular device set in the PDI. In many cases, VL components may include direct accesses to vendor-specific connectivity protocol stacks and media control components.

Physical Device Interface (PDI)

Provides a physical, or driver-level interface to actual physical devices assigned to the control of the VCO implementation. The PDI inherits virtualization functions from the VL to provide a rigidly compliant implementation of a device control interface used by the VDI directly to provide support for its logically defined tasks. PDI implementations include direct accesses to vendor-specific connectivity protocol stacks and media control components.

Encapsulated Devices

The encapsulated devices in a VMCS typically include a network interface unit and a host of related

media control devices. The connectivity protocol stack refers to the software layers necessary to provide services defined for the OSI Transport Layer; that is, it must ensure the reliable transfer of messages between end users. Media Access Control must contain drivers enabling physical operation of all devices relegated to VCO control, as previously specified in the section entitled DEFINITIONS. The types of devices likely to be incorporated into the VCO design, will be some variation upon those described to manage audio, video, images, and binary data, as specified in the section entitled I/O Equipment.

Component Interactions

There are well-defined modalities of interactions between VMCS components. The VDI makes direct use of PDI members to invoke the services of physical devices in its mission to fulfill VCO Client requests for services. The PDI, in turn, calls functions in the connectivity protocol stack and in the Media Access Control layer. The PDI calls member functions in the VL to provide the mapping, translation, and formatting necessary to coerce the low-level services to resemble those logically specified by the VDI. As they occur in real-time, events generated by the connectivity protocol stack and Media Access Control components are added to a general VCO event queue. A dispatcher in the VCO removes events periodically, synchronous with the operating system scheduler. An event processor in the VL responds to events as they are dispatched to it, invoking other VCO components as needed. Some of these events require that the client application be notified of their occurrence, if the client has so arranged.

VCO ARCHITECTURAL OVERVIEW

FIG. 3 depicts a functional block diagram of a VCO, with components partitioned according to the VCO layer where they reside. This section provides a high-level view of the VCO sub-components' functional organization. Subsequent sections pursue a more rigorous examination of the constructs themselves, here only topically considered. It is more important to note that the VDI, VL, and PDI sections labeled "Device/Protocol Controllers" are to be considered as layer-specific overlays of the same groups of components. The groups in the VDI section "Device/Protocol Controllers" illustrate the logical definitions for each of ten distinct functional categories; corresponding groups in the VL and PDI describe the identically functional categories, but at a different level of software abstraction. All components in the VDI section are fully reusable, device-independent software components. Those components in the PDI are vendor-specific and implementation-specific while those in the VL are mostly device-independent, but may require some implementation-specific coding to support wide variations in the underlying device control software.

Software components in the VCO are physically divided into very specific object entities, each of which much interact with those adjunct and adjacent. A set of related functions and data structures combine synergistically, are referred to as a controller. Such entities are the basic functional units of the VCO in that they form discreet functional "parts" that control and/or manage a well-defined set of tasks.

Summary of VCO Architectural Overview

The VCO is a single binary software object that encapsulates all of the hardware and software components

necessary to implement a fully Virtualized Multimedia Connection System. Virtualization of the services provided by disparate connectivity and media control systems is rendered according to three-layer progressive abstraction strategy that relies upon object-oriented software technology for both its design and implementation.

- A device-independent, reusable software layer call the Virtual Device Interface (VDI) is created to provide a detailed logical description of a virtualized multimedia connection service. It has as set of public member functions that define its interface to applications and other invoking software modules. Included in this layer are a series of software controllers that are specifically here located (in this logical layer) to embody the larger part of the system's software implementations in a layer that would be directly propagated to new system implementations, wherefore to facilitate the creation of a highly reliable, reusable, portable modules.
- A mostly reusable, and predominantly device-independent intermediary software layer called the Virtualization Layer (VL) is created to assist in the process of translating and mapping the functions of various device control mechanisms employed by the underlying connectivity and media control sub-systems, to the logically defined virtual representation required by the VDI.
- A device-dependent layer called the Physical Device Interface (PDI) interfaces directly to the device control sub-system to provide a

physical implementation to the service requested by the VDI. The PDI makes use of virtualization functions in the VL to coerce the particular message and data output formats of vendor-specific device control components, to a structured format integral to the VCO design.

- Audiovisual and binary data streams in the PDI sub-system are represented as Media Control Objects (MCO). A list of these objects is maintained by the PDI so as to present the VCO with a pool of available media device and signal resources. According to the media type, and its direction of flow in the system, these MCOs are classified accordingly as either an audio, video, image, or binary data type. They are further characterized as a signal from the remote station, to the remote station, to a local output device, or from a local output device. Operations performed on these objects modify their properties and modalities of interaction, so as to provide the VCO with a logical software switching mechanism for its encapsulated media control device inputs and outputs.

Virtual Device Interface

The device-independent portion of the VCO, is a fully reusable entity that is created once, and then propagated to all VCO implementations. It contains the definition of all the VCO public member functions accessed by clients. These members are collectively referred to as the Software Control Interface (SCI), which itself includes a number of other VCO control

mechanisms whose functionality extends well beyond simple calls to members. The VCO event notification mechanism and terminal stream I/O manager are integral to the VDI, as are ten controllers related to implementation of the services requested through calls to SCI members.

Software Control Interface Functional Groups

These groups of "control members" contain the member functions used by clients to invoke VCO services. Each relates to a set of public VCO member functions used to access a specific class of well-defined operations.

- **Event Notification Control Members** enable the creation, deletion, and configuration of "Notifier Objects" (NO) that may be programmed to trigger on the occurrence of any one of a defined set of VCO events. When a Notifier Object triggers, it make a function call to a special member function of a specified object, and thereupon conveys information concerning the event that occurred.
- **Configuration/System Setup Control Members** enable VCO configuration information to be saved to, or retrieved from backing store.
- **Terminal Services Control Members** enable writes of text messages to VCO terminal output port, and command input through the VCO terminal input port. They also allow the VCO terminal output port to be attached to various output devices.
- **Network Session Control Members** enable establishment of a network session with a remote station. They include members to invoke initialization and shutdown of the physical device control sub-system, as well

as to place point-to-point and multipoint calls.

- **Audiovisual/Data Signal Switching Members** enable signals to/from the remote station, and to/from transducers to be manipulated, combined, and interconnected in various combinations.
- **Binary Data Object Exchange Control Members** enable buffers, files, and other binary data entities to be transferred between the local and remote stations.
- **System Information Store/Retrieve Control Members** enable access to VCO parameter blocks containing information about their encapsulated devices, current call states, protocol states, configuration, and control/monitoring context.
- **Protocol Management Control Members** enable direct manipulation of the H.320 protocol whose implementation is integral to the VCO. Allows advanced operators to perform mid-level operations to configure the VCO, and any connected remote station, according to the procedures of H.320.

Notification Controller

This notification mechanism is used both internally by the VCO, and by client applications that wish to monitor, or respond to specific system events. A Notifier Object is created, and programmed to respond to any subset of the cataloged VCO events. If the event occurs, a special notification member in a target object is called and passed information regarding the occurrence of the event. The Notification Controller refers to all of the member functions and member data necessary to

implement the notification mechanism in each VCO. It contains three distinct parts that operate both independently and together, depending on the notification task at hand.

- **Notification Triggering Mechanism** is responsible for determining exactly when a Notifier Object should trigger, by examining its list of triggering events when one such event occurs. If the Notifier Object is set to trigger on the event, it is activated to initiate its trigger sequence, thereupon informing a specified software object that the event has occurred; it passes parameters during a call to a notification member function.
- **Notifier Object List Manager** is responsible for creating, deleting, modifying, and querying a database containing all Notifier Objects for the VCO.
- **Linked Notifier Object List** is a doubly linked list of all Notifier Objects for a VCO, and it is maintained by the Notifier Object List Manager.

Terminal Stream Controller

This terminal stream mechanism is used both internally by the VCO, and by client applications that wish to direct streams of text messages to a configurable set of terminal output port destinations that may include files and system character devices. Alternately, streams of text messages directed to the VCO terminal input port are interpreted as VCO commands and invoke standard VCO operations. This Terminal Stream Controller refers to all of the member functions and member data necessary to implement the terminal stream I/O mechanism in the VCO.

It contains three distinct parts that operate both independently and together, depending on the terminal operation.

- **Terminal Stream I/O Device Controller** is responsible for processing text messages sent to the VCO terminal input and output ports, in addition to managing the list of I/O devices.
- **Text Command Encoder/Decoder** is comprised of two separate components: the encoder portion converts calls to the SCI into text-token string command representation used for remote VCO control, and the decoder parses an input stream of such text-token string commands and makes calls to the SCI as indicated by their format.
- **Linked Terminal Stream I/O Device List** is a doubly linked list of all terminal I/O device objects that describe the identity and status for all character stream files, and devices to which text messages sent to the VCO terminal output port are written.

Device/Protocol Controllers

This group of controllers serves to support the implementation of the logical operations invoked by calls to SCI member functions, as well as providing implementation of operations necessary to the establishment and maintenance of a connectivity session. The implementations of services in this VDI component must be only that portion of the given operations that can be supported in a fully device-independent manner.

- **Object Construction** refers to all procedures and data structures involved in the construction of the VCO, including

initialization of all object software components that do not control devices. Provides direct support for operations invoked by construction of the VCO by a client application.

- **Object Destruction** refers to all procedures and data structures involved in the destruction of the VCO, by its client application, including shutdown of all object software components.
- **Configuration/System Setup** refers to all procedures and data structures involved in configuring the VCO according to its profile, previously saved on a backing store device. Also includes support for specialized audio, video, image, and binary data equipment setup utilities. Provides direct support for operations defined for the SCI Configuration/System Setup Members.
- **Binary Data Object Exchange** refers to all procedures and data structures involved in the transfer of named binary objects, such as system files, to and from the remote station. Provides direct support for operations defined for the SCI Binary Data Object Exchange Control Members.
- **Network Call State** refers to all procedures and data structures to support a call controller compliant with that described in the section entitled Visual Call Progression, and consequently function to establish and maintain the connectivity session with the remote station. Provides direct support operations for internal VCO connectivity sessions.

- **System Information** refers to all procedures and data structures involved with storage, retrieval, and maintenance of the various VCO parameter blocks, and their associated tables and lists. Provides direct support for operations defined for the SCI System Information Store/Retrieve Control Members.
- **Capability Exchange** refers to all procedures and data structures needed to support a structured exchange and comparison of device capabilities, as described in the section entitled Visual Call Progression. Provides direct support operations for internal VCO connectivity sessions.
- **System Exception** refers to all procedures and data structures involved in handling fatal errors that may occur in the VCO. Provides direct support for operations defined by SCI VCO Settings Members not shown in the drawing.
- **Media Control Object Access** refers to all procedures and data structures involved in accessing the object-based device control system implemented in the PDI. Provides direct support for operations defined for the SCI Audiovisual/Data Signal Switching Control Members (that are not shown in the drawing.)
- **Protocol Mode Negotiation** refers to all procedures and data structures involved with establishing a common set of parameters between the local and remote stations as described in the section entitled Visual Call Progression. Here also lies support for operations that will affect a specific conference profile as specified by the VCO's

configuration parameters (or as set by the client application); includes support for operations essential to internal VCO network session implementation.

Virtualization Layer

Contains all procedures and data structures used to convert vendor-specific formats to those defined by the VCO. This layer also contains a number of controllers that are not always able to be implemented in a device-independent fashion, as some cognizance of vendor-specific peculiarities is frequently necessary to create a functional virtualization service layer.

Event Controller

Assigned responsibility for modulating the flow rate of device and software-generated events, in and out of an event queue. The queue acts a storage repository to record both the progress of operations carried forth by the VCO, as well as reports of new states and modes assumed by its underlying real-time device control sub-system. In general, devices and VCO controllers perform a mutually exclusive addition of events to an infinite sized event queue, in real time. A dispatcher removes them at a regular periodic rate, and disseminates them to an event processor, and to client applications desirous of knowing that one (of a particular set of events) has occurred.

- **Event Processor** provides a structured mechanism for the VCO to respond appropriately to events generated by its device control sub-system. It maintains a number of feedback loops and procedures that are critical to enabling the VCO to adequately monitor and control its

connectivity session without assistance or monitoring by the client application.

- **Periodic Event Dispatcher** checks the VCO event queue at regular intervals to determine if it contains any events. If there is at least one event in the queue, the dispatcher removes it (a single event) and invokes the Notification Controller to trigger any Notifier Objects that may be configured to respond to that event.
- **Infinite FIFO Event Queue** provides a sequentially-ordered cache for all events emanating from the device control sub-system or for events generated by software components within the VCO. It operates according to the first-in-first-out algorithm so that the original sequence at which the events occurred, is preserved through the event processing stage. Since the queue is constructed as a linked list of event records, it is capable of growing to a size limited only by the availability of system memory; thus events are never lost in any VMCS.
- **Critical Section Handler** provides a mechanism to add events to the event queue in a mutually exclusive fashion. Also, various VCO operations that could result in resource contentions and deadlocks can utilize this component.

Linguistic Controller

Assigns textual labels to VCO events, states, operations, and a host of other functional entities, in a way that describes them using plain language. The role of

this controller is to enable VCO components to report the status of their operations in a readily-understandable format, rather than by the encoded strategies typically employed in computer systems. Various VCO controllers derive descriptive text messages from the labeling facilities offered here, and then send them to the VCO terminal output port.

- **Event Labeler Mechanism** provides functions to return string labels for various VCO events of all types. In addition to returning text labels for the standard VCO notification events, this controller supports labeling for the generalized class of VCO events that includes all of the enumerated states, constants, modes, and string tokens representations of the arguments used by the Media Control Object Device Control Mechanism. The command encoder/decoder mechanisms also rely on the services offered by this component.
- **Object Component Labeling Mechanism** provides labels for VCO objects and constructs that are used to report the current location of execution, or processing, in terms of the names of actual VCO components invoked. Contains the names of the VCO classes, controllers, an mechanisms that are used to formulate precise reports for debugging, diagnostics, and general troubleshooting of VCO components.
- **Event and Object String Label Database** contains tables of text tokens and string messages accessed by the event/object component labeler mechanisms.

Device/Protocol Controllers

The Device/Protocol Controllers for the VL and PDI layers, as shown in FIG. 3, represent the contribution of each layer in producing the virtualized multimedia connection services logically defined in the corresponding VDI controller (that occupies the same physical location on the drawing relative to the group of Device/Protocol Controllers). This group of VL controllers serves to support the implementation of the device-independent procedures and operations supported, at their highest level, by the corresponding controllers in the VDI. As the VL controllers are specifically Virtualization Layer components, they serve to support the implementation of VDI controllers by providing any necessary coercion of data format, command syntax, or interface method, from a vendor-specific modality, to that defined for the VDI. In many cases, the VL supplies a standardized function supporting a like standardized VDI function, but the embodiment of that in the VL is fully intended to be implementation-dependent.

- **Software Component Load/Initialize** provides implementation-dependent mechanism to load and initialize all supporting software modules, libraries, and device drivers necessary to VCO operations. Called by VCO constructor, the VCO construction fails if all necessary components are not present or fail to load. No devices of any sort are initialized or accessed explicitly here, except to determine if they are installed.
- **Software Component Unload/Shutdown** provides implementation-dependent mechanism to unload all supporting software modules, libraries, and devices drivers. Does not shut down

devices directly, but does free all resources associated with VCO.

- **Configuration Information Access** provides any necessary mapping of configuration information from/to the format used by the backing store method. Typically reads from/writes to a string-based initialization file and translates the string token format to that used by the VCO configuration parameters.
- **Data Exchange Syntax Mapping** provides mapping of file and record formats to that utilized by the connectivity protocol stack to that used by the host operating system. Examples include conversion of buffers and files to/from packet formats.
- **Call Event Mapping** provides translation of vendor-specific connectivity protocol stack representation of call and line states to those used by the VCO call controller.
- **System Information Mapping** provides translation of various vendor-specific representations of system information to/from those used by the VCO, and enables translation of Media Control Object Device Control operations to H.221 device modes. Specific translation examples include call destinations, and media device states.
- **Capability Mapping** provides translation of local and remote station H.221 capabilities emanating from the network protocol stack to those used by the VCO. Includes mapping of H.221 capabilities to Media Control Object capabilities.

- **System Exception Mapping** provides translation of various fatal errors from vendor-specific components to standardized VCO exceptions.
- **Media Device Driver Access Mapping** provides translation of Media Control Object Device Control operations to Media Control Interface string commands that are presented to the Media Access Control layer.
- **Protocol Mode Mapping** provides mapping of H.221 bit-rate allocation signal (BAS Code) indications used by the connectivity protocol stack (or emulated) to the device-independent representation used by the VCO.

Physical Device Interface

Contains all procedures and data structures used to interface device drivers and operating system resources directly. All implementations of functions are assumed to be device-dependent, and it is the principle objective of this layer to locate some form of physical, or emulated support to realize those operations logically defined in the VDI. If the PDI cannot provide, or even emulate, a service specified by the VDI, then it must return a result code indicating that it is not capable of doing so.

Device/Protocol Controllers

The Device/Protocol Controllers for the PDI layer represent the contribution of this layer in providing the physical interface component of the virtualized multimedia connection services logically defined in the corresponding VDI controller. This group of PDI controllers serves as the physical implementation of the corresponding operations managed by controllers in the VL. As the PDI controllers are specifically physical

layer components, they support the implementation of VDI controllers by directly accessing device drivers and vendor-specific library software components provided by third party OEMs; in the most efficient designs, the PDI may be implemented as an actual device driver, directly controlling hardware.

- **OEM Support Libraries and Drivers** provide specialized functions, such as image processing, digital signal processing, data port interface, and system diagnostics, that are vendor-specific and usually hardware-specific.
- **Configuration Information Backing Store** is the physical device that stores the VCO configuration information. Usually a file on disk and/or some form of non-volatile memory device.
- **Connectivity Protocol Stack** is the actual interface to the network, and includes all associated hardware and software components. The network service is presented to the VCO session components and transducers at the level of the OSI Transport Layer, whose responsibility it is to ensure reliable transfer of messages between end users. The Network and Data Link layers must be available, in some form, below the Transport Layer, and a physical or emulated network interface unit must be present and detectable.
- **Media Access Control** provides physical device control mechanisms for input and output transducer devices in the system. This driver complement is used directly to implement the PDI's defined set of device control interface

functions used directly by the VDI, as well as by the Media Control Object Device Control Mechanism.

- **Device Capability List** resides in the PDI as the master list of the all of the VCO capabilities. It is stored as an array of device-independent capability constants that reflect the range of H.320 (H.221) capability BAS codes.
- **Device Mode List** resides in the PDI as the master list of the all of all possible device modes for any H.320 compliant station. It is stored as an array of device-independent device mode constants that reflect the range of H.320 (H.221) mode command BAS codes.
- **Media Control Objects (MCO)** are constructed and maintained by the PDI, and presented to the VDI, as a method to control all signals to and from remote stations and transducers under VCO administration. The PDI interacts directly to support the device-dependent implementations that underlie these device-independent representations of audiovisual/data signals and devices as discreet system objects, each possessing a structured set of properties and well-defined operations.

VCO SOFTWARE ARCHITECTURE

System Dynamics

It is useful to consider the VCO as a mechanical device, much like a spring-driven mechanical clock movement. Each VCO is a self-contained machine of interlocking parts, with a system timer interrupt advancing its works by the increment and released in

balanced measures that bring stability and smoothness of operation to the system's "top" end. The VCO's analogous "unwinding" takes place with the literal precision of a clock's escapement mechanism, in that the system timer serves as the regulatory entity releasing a steady pulse of queue-stored activities from the device control sub-systems, to a client application. A Virtualized Multimedia Connection System presents not only an idealized control mechanism to applications, but also idealizes the rate and content of the system's responses to these controls, without ever losing a system event, state change, mode change, exception or interrupt request. The overall VCO design is consistent with that of multi-threaded, queue-based device driver designs that account for a significant portion of the dramatic gains in reliability and performance seen in the use of 32-bit, multi-tasking operating systems such as IBM's Operating System/2 (see H.M. Deitel, M.S. Kogan, The Design of OS/2, Reading, Massachuetts, Addison-Wesley, 1992).

Notes On Drawing

FIG. 4 depicts an a detailed diagram of the major components and control flows of the Virtual Connection Object software architecture. The Device/Protocol Controllers shown are the same as those shown in FIG. 3, but the purpose in this drawing is to illustrate the direction of control flow through them, rather than to describe what they are. This discussion will examine the VCO in terms of its specific software mechanisms. To clearly show the functionality of individual VCO components in the two-dimensional drawing, it is necessary to alter their exact locations in the layering model from a perfect vertical orientation, to one more suited to revealing component interactive pathways.

Summary of VCO Software Architecture

The software architecture of the VCO is can be described best in terms of two major functions: (a) controlling the multimedia connectivity sub-system it encapsulates, and (b) responding to events emanating from it. What ensues is a brief overview of outstanding features that characterize the dynamics of the VCO software model.

- The Software Control Interface (SCI) contains public member functions that enable a client application to access a consistent and logically-defined multimedia connectivity service to create a live audiovisual connections to a remote station. Calls to these members invoke the various Device/Protocol Controller services in the various VCO layers until some physical operation is eventually performed, the result of which is translated to logical, standardized result prior to being returned to the caller.
- Each VCO contains a general repository for all system information that is continuously updated to reflect the current states of all controllers and devices.
- Every VCO has standard input and output terminal ports that function much like the standard input and output devices in operating system shell console devices. All text message sent to the VCO terminal output port are written to a list of terminal I/O devices maintained by a Terminal Stream Controller. This controller also accepts text commands written to the terminal input port and decodes them, translating them to SCI

calls. An infinite FIFO event queue accepts the addition of events asynchronously, in real-time, by device control software, and by software components in the VCO that generate their own particular events.

- An Event Dispatcher runs synchronous with the operating system scheduler (does not preempt the regularly scheduled time slices) to remove events from the queue at a constant, periodic interval and disperse them to a Notification Controller.
- The Notification Controller examines the dispatched event, dispensing notification information about the event via Notifier Objects that send the indication, and all associated information, to an event processor in the VCO (if the event is from a device), or to a client application object, depending on the Notifier Object's configured destination object.
- The VCO Device Event Processor invokes procedures to respond appropriately to events emanating from the device control sub-system, so as to maintain a connectivity session with the connected remote station. Call control, capability exchange, and various protocol operations are driven by the event processor, as is the issuance of text messages, describing all system activities, to the terminal output port. Most network events that require attention from the application in similar multimedia connection systems do not require such attention by VCO Clients; specific intelligences in the Device Event Processor component better direct them to

invocation of specialized internal procedures more suitable to such tasks, by their very designs -- feedback loops often found in the application layer, reside in the VCO proper, alleviating the application developer from implementing sensitive protocol-specific modules.

- Ubiquitous throughout the VCO's controllers are reporting mechanisms that formulate detailed text messages describing the current state, mode, process, subroutine, class name, or event that is currently most relevant, and sending this string to the terminal output port. An Event and Object Labeler working in conjunction with a string database, has features that can assign text names and terms, in addition to maintaining a text-token definition of the VCO text command set.
- VCOs can link together across a connection to control or monitor the activities of the other station. This concept, referred to as attachment, utilizes a bi-directional text message stream to enable interconnected VCOs to share commands and events. Depending on the negotiated configurations of the "attached" VCOs, calls to member functions in the SCI of one, are encoded into text commands by the Command Message Encoder, and transmitted to the other station. Upon receipt, the Command Message Decoder translates the text commands back to SCI calls. Correspondingly, events queued at one station, as well as results of operations, are encoded by the Event Message Encoder, and transmitted to the other station. Upon

receipt, the Event Message Decoder translates the text event messages back to events and adds them to the event queue, where they are dispatched along with a station identifier.

Software Control Interface

The Software Control Interface (SCI) consists of the VCO's public member functions and protected data that allow client applications to control the VCO. Any request for service using the SCI must pass a number rigorous examinations designed to reject any possibility of damaging the encapsulated sub-system. A client attempt to access a VCO member function results in an immediate set of verifications to determine if VCO is available for use, and if so, whether the context of the request is valid. For example, most operations require that the device control sub-system first be fully initialized. Arguments are checked for validity, and a text message describing the request is sent to the VCO terminal output port for tracing and monitoring purposes. If all conditions have been met, execution continues to the next level; progressing typically to a further request to the PDI. Rejected requests are turned away abruptly, but the client is compensated for his diligence, with a fair accounting of the dismissal; every operation returns a concise result code.

Member Functions

Member functions to invoke the services of the VCO are partitioned into several distinct categories. The members and their arguments are highly structured and flexible in the variety of ways they may be utilized. They are easily mapped to a text-token command set. All of the members of this interface are available from one constructed VCO, and each optimized to best support the

likely use of it, rather than providing similar access methods for all command permutations equally. The pathways through the VCO layers and components are unique for each functional group, and are summarized briefly:

- **Event Notification Control Members** make calls to the Notifier Object List Manager to create, delete, and configure, and trigger Notifier Objects in the Linked Notifier Object List. Calls to trigger notification invoke the Notification Triggering Mechanism directly.
- **Terminal Service Control Members** make calls to the Terminal Stream I/O Device Controller to add and remote devices from the Linked Terminal Stream I/O Device List. Calls to send text messages to the terminal output port are made through this object, as are calls to send text command message to the terminal input port for decoding and execution.
- **Configuration/System Setup Control Members** make calls to the PDI to store and retrieve configuration information from some physical backing store device.
- **Network Session Control Members** make calls to the PDI to initialize and shutdown the encapsulated multimedia connectivity sub-system, as well as to initiate and terminate sessions with a remote station.
- **Audiovisual/Data Signal Switching Control Members** make calls to the PDI to manipulate audio, video, image, and data objects that comprise the Media Control Object Device Control System of the VCO.

- **Binary Data Object Exchange Control Members** make calls to the PDI to transfer binary data objects and data buffers to and from the remote station.
- **System Information Store/Retrieve Control Members** make calls that access the VCO system information repository in a structured manner; a manner which does not allow for any direct modification of VCO data structures.
- **Protocol Management Control Members** make calls to the PDI to directly set H.230 device modes, send capabilities to the remote station, and perform other protocol related tasks.

Terminal Service

The VCO terminal service has two main pathways: into the VCO through the terminal input port, and out of the VCO through the terminal output port. Alternately, when the terminal ports are considered as standard input and standard output devices for the VCO, it is sensible to view the two modalities as a stream "to the terminal" and "from the terminal".

- **Output Streams to VCO Terminal** originate from the VCO itself, or from client applications. Messages "to the terminal" are directed to the terminal output port and dispersed, via write operations, to output devices (attached to the terminal output port). Summary, "To terminal" is synonymous with "to text output devices."
- **Input Streams from VCO Terminal** originate from outside the VCO, typically from a dumb terminal or client application wishing to control it with text commands. SCI calls

sending text messages as if "from the terminal" are directed to the terminal input port and interpreted as input text command messages, as if input from a keyboard. Summarily, "From terminal" is synonymous with "from the standard text input device."

- **Terminal Stream I/O Device Controller** maintains a linked object list of output device objects to which the text message sent to the terminal output port are written. All text messages sent to the terminal output port are written to every device cataloged by the linked object list. Output devices include system files, character devices, and Notifier Objects created for the transmission of text message to objects.

System Information Handling

System information is fully protected from direct external access the VCO, though all internal components can access it directly. Clients must us specific member functions to get a copy of the data, and use other members to affect changes to it through structured operations. Internally, all system parameters are fully accessible to all components derived from the VDI.

- **Configuration Parameters** store the current copy of VCO configuration and system setup information that is read from backing store at VCO construction time. Contains information about the network service available, and all system defaults. The parameters can be modified and written back to backing store at any time.
- **Device Parameters** store all settings and parameters related to the

audio/video/image/data devices installed in the sub-system, and retain handles to the current source, destination, input, and output signals affected by the Media Control Object Device Mechanism.

- **Call Parameters** store all of the current call and line states, including those for multipoint calls. Maintains records about other stations in the conference.
- **Protocol Parameters** store all current transmit and receive modes for all the various data types.
- **Control Prameters** stores all information related to maintaining the remote station control mechanism for any attached station.
- **Monitor Parameters** store all information related to maintaining the monitoring access for any attached station.

Notification Mechanism

The VCO notification mechanism conveys an indication that a particular event has occurred by activating, or "triggering", a notification entity referred to as a Notifier. Maintained in a list internal to the VCO, Notifier Objects are created specifically to call a member function of some appropriately enhanced class object, somewhere in the system (within or without the address space of the VCO) when triggered. Upon the VCO event dispatcher's presentation of an event to the Notification Controller, Notifier Objects in the list are systematically examined, and potentially triggered, according to a specialized modality of governance that considers the relationship of the outstanding event type (among other parameters) to the triggering profile of the Notifier Object itself.

Notifier Objects (NO)

These objects comprise the basic indication unit of the VCO notification mechanism. Each NO is an instance of a class object that may be created by a VCO client, or the VCO itself, and configured to "trigger" in response to the occurrence of any one of a set of VCO events. Each NO is associated with a specified Notification Receiver Object (NRO) to which the trigger response is directed. When an NO triggers, it makes a function call to the associated NRO member function assigned to the task of handling notifications. Passed to the NRO is an event identifier, and a number of qualifying arguments that describe the context of the event's occurrence. There are two mutually exclusive modalities for any NO; they can be configured to respond to any event, or configured to respond only to events emanating from VCO-encapsulated devices.

Notifier Management

This task is handled by the Notifier Object list handler. This component adds to, removes from, and maintains the integrity of the Notifier list. It also provides members for configuring Notifier Objects with new trigger response profiles, as well as to allow them to be enabled, disabled, and examined for statistical information. In the VCO, the notification mechanism is supported by a component that manages the list of all active Notifier Objects, and a triggering mechanism that determines as to whether or not an individual Notifier should trigger, based upon the occurrence of an event to which it is configured to respond.

Triggering Mechanism for Notifier Objects

This mechanism conditionally determines as to whether or not a Notifier Object should trigger, based

upon a specific algorithm. Notifier Objects can be configured to respond to events that emanate from the device only, so as to direct their trigger response to the VCO Device Event Processor. Through this method, the VCO uses Notifier Objects internally to create a direct pathway for device events (added to the VCO queue by the device), to be processed in the VCO Device Event Processor exclusively. This distinction serves to prevent an infinite loop that would result if the VCO processes an event from a device, and then reissues (requeues it) it so that client applications can be subsequently notified of the same event -- the same event would be dispatched back to the event processor again and again if reinterpreted as another occurrence of the same device event. Thus a distinction must be made between the original device event and a processed derivative of that same event intended for dispatch to the client application. Notifier Objects configured to trigger on events directly from devices will only trigger on events directly from devices. When the event processor reissues the event from the device, it marks it to indicate it is not directly from a device, and it can no longer trigger the Notifier that would send it to the Device Event Processor.

Notification to Clients

VCO Client applications may request notification of a combination of VCO events by creating a Notifier Object and configuring it to trigger when any single event in the combination actually occurs. Once created, the Notifier Object conveys event notification to an application object set to that purpose. The object that receives notification is referred to as a Notification Receiver Object (NRO).

- **Notification Receiver Objects (NRO)** receive function calls from an entity in the VCO that is created specifically to direct notification of the occurrence of an event to them. A class object can serve as an NRO if it contains a special Notification Receiver Member and an attendant thunk. The thunk is a globally accessible function that is created to receive notification from the VCO and redirect that notification to the NRO's Notification Receiver Member. In this way, a VCO can deliver a notification message to a class object which can then directly access other class members and member data that would not be available if the notification was to a non-member function i.e. had to access class members with a special class pointer.
- **Notifier Triggering Profiles** refer to the set of events to which the Notifier Object is configured to respond. Notifier Objects are "triggered" to notify their associated NRO when one of these configured events occurs.

Event Handling Mechanism

The event handling mechanism consists of three concurrent, but distinctly separate processes. From the perspective of any single event, these processes occur in a serial fashion. First, events are added to the trail of the VCO event queue by various VCO components. Next, a concurrently executing event dispatching process periodically removes an event from the queue. Finally, the event dispatcher sends the removed event to the Notification Controller where it triggers Notifier Objects configured to respond to events of this

particular type. If the event is a device event, and the Notifier is configured to respond to device events, then its Notification Receiver Object receives notification that the event has occurred. If the event removed by the dispatcher is some derived event, it may trigger notifications to client application, or any other Notification Receiver Objects targeted by a Notifier responding to that event.

Device Events

These events are generated directly by a device in the encapsulated sub-system; a situation that potentially requires some kind of immediate responsive action. They are the primary responses from VCO devices and emanate from network and media control driver software components.

Derived Events

These events are generated by a VCO software component, and are derived from an action or logical state, not directly emanating from a device. They are the secondary responses from the VCO itself, often resulting from the processing of device events.

Event Processing

The task of event processing includes the handling of both Device Events and Derived Events. Device events can only trigger Notifier Objects configured to respond to device events, thereby enabling a particular Notifier Object to function as a dedicated device event notifier. Events entering the Device Event Processor are typically line state changes, device mode changes, and indications from the remote station. These events drive protocol and session management routines during processing, which in turn generates derived events such as the call state

changes and Media Control Object Device Control parameter changes. These derived events are queued for subsequent dispatching to client applications; these secondary occurrences are treated differently from primary events in that they will not be handled by the Device Event Processor.

Event Dispatching

Dispatching of events occurs periodically at a programmed rate that may be adjusted dynamically at run-time. Typically, dispatching five events per second is sufficient to present the client application with a manageable stream of events. The event dispatcher is driven by a system timer interrupt. If the queue is available for mutually exclusive access, it is checked to determine if there are events in it. If there is at least one event in the queue, one event is removed in a critical section, and the queue is released to the system. If mutually exclusive access is not immediately available, or there are no events in the queue, the dispatcher yields. Otherwise, the removed event is next presented to the Notification Controller where any Notifier Objects in its list, sensitive to the event, are triggered so as to disperse the event throughout the system.

Event Queuing

Queuing of events occurs sporadically, when an event is generated by a VCO-encapsulated software or hardware component. Frequently occurring during hardware interrupt cycles, queuing is carefully optimized to require very few cycles and little resource allocations. A short blocking operation may be necessary during dispatching to gain mutually exclusive access to it. The event is added and the queue released back to the system.

If the attempt to add an event to the queue fails, a fatal error (VCO Exception) occurs and the VCO is permanently disabled.

Exception Handling Matters Mechanism

The VCO exception handling mechanism is not shown in FIG. 4. Exceptions in the VCO are conditions that are deemed fatal errors from which the system cannot recover without risking a system crash. The underlying design principle to VCO exception handling is that system crashes result most often from continued use of a destabilized system component, from attempts to recover from destabilized conditions, and from the failure of a destabilized system to acknowledge its "time-bomb" status. In accordance with an overriding concern for system reliability, VCO exceptions generate reports, and a subsequent disabling of the VCO. Neither the VCO, nor its concomitant support software, is accessed until the VCO is destructed. Once disabled, all calls to the SCI return immediately with a result code indicating that the VCO has been disabled. For continuous use applications, the risk of system crash is rendered unlikely, and the system with one destabilized sub-system likely remains of sufficient health to invoke a resident backup system.

Fatal errors occurring internally to the VCO generate an internal assertion failure that invoke a recovery routine that proceeds to execute a reverse stack trace. The trace searches for markers pushed onto the VCO stack during calls to a special member function called upon entry into the VCO. Every VCO entry point is guarded by a call to a function that returns a result code indicating whether or not the VCO is enabled or disabled. Upon locating the address of the instruction pointer (on the stack) at the execution point of this status call, a result code indicating that the VCO is disabled is pushed

onto the stack, along with the address of the entry point. A return instruction is executed to bring the calling thread back to its original point of execution just prior to calling the VCO status member (which will indicate that the VCO is "unavailable" when the status call is re-executed), and to the impression that it has been turned back at the door due to a preexisting disabled state. The calling thread is without cognizance of the fatal error it unwittingly generated by its venture inside the VCO, and finds the VCO is simply unavailable for continued use. This technique of shutting down the damaged VCO, accompanied by transparent error handling (from the perspective of invoking applications), maintains system integrity until the host system can risk a recovery operation.

Exception Handler Modalities

A variety of VCO exception handling modes can be invoked incrementally by specifying any combination of the following modality flags. These modalities are additive, and affect the way in which reporting of the exception is handled.

- **Debug** modality flag, if set, specifies that detailed debugging information (in a message box) will be displayed at the time of exception. This mode is intended for use during system development where proprietary information will be displayed.
- **User** modality flag, if set, specifies that general "user" information (in a message box) will be displayed at the time of exception. This mode is intended for use in the field after the product is released.
- **Term** modality flag, if set, specifies that exception information will be sent to the VCO

terminal output port for display on terminal output devices.

- **Notify** modality flag, if set, specifies that the exception will generate an exception event and trigger Notifier Objects prior to disabling of VCO.
- **Abort** modality flag, if set, specifies that the exception will abort all VCO operations, after reporting the exception, and disable the VCO.

Event and Object Labeling Mechanism

This mechanism provides function calls that convert indexes to strings. It relies on a collection of string tables, whose string element positions reflect the value of the indexes. The string tables can be loaded from resource files that contain text in the appropriate language.

Labeling of Events, Enumerated Values and Result Codes

Retrieving a text label to describe a VCO event is accomplished by presenting a VCO event identifier, result code, or standard enumerated constant to the appropriate member function. A pointer to a static copy of a null terminated label is returned, and is typically used to formulate messages for terminal output, and by applications to display states, modes, and operations taking place in the VCO.

Labeling Objects

Retrieving a text label for a VCO software object is accomplished by presenting a pointer to the object to a member function. A pointer to a static copy of a null terminated label is returned, and is typically used to formulate messages for debugger and trace window output.

Used specifically for debugging, diagnostics, and troubleshooting.

Event and Object String Label Database

A memory resident database in the VCO contains tables of string records for all of the VCO enumerated constants, and the VCO command set. Reference databases containing object pointers and labels are created at VCO construction time.

Media Control Object Device Control Mechanism

Designed to facilitate the manipulation of signals as distinct objects, this device control method is intended for full implementation as a graphical desktop media control system, supporting a drag and drop signal switching model. Each object, representing a specific signal type, has a standardized set of properties, and well-defined modes of interaction. The physical structure of the Media Control Object Device Control Mechanism represents each signal as a software class object, and all signals currently available in the system as a linked list of such objects in the VCO DEVICEPARAM structure. Media Control Objects are created and deleted as signals become available, or unavailable, depending on connection state and device availability.

VCO signals are identified according to their type and direction. The possible types of signals include audio, video, image, and binary data. The possible directions include input from the remote station, output to the remote station, input from a local transducer, and output to a local transducer. Member data in each Media Control Object describe the current states and modes related to the signal. Member data in each Media Control Object describe the current states and modes related to

the signal, and member functions invoke physical operations in their host devices.

It is possible to determine if a given Media Control Object is capable of specific operation by setting the query flag on the SCI member function to control them. In this manner, the client can test an operation without invoking it. A special function to return the capabilities of a setting is also available, and a list of settings constants (or parameters for the setting) is returned. For H.221 capabilities, related to the multimedia interconnection protocol, the VCO parameter block maintains an updated listing. There is a very close logical mapping between H.221 capabilities and Media Control Object capabilities -- no H.221 video mode means that there is no motion-video available -- and it is likely that the very existence of a Media Control Object indicates that most of the operations for that signal type are supported to some degree. It is often sufficient to let a client attempt to set the mode, and report that the system is incapable, than to constantly monitor and maintain a local capability listing.

Device/Protocol Controllers

The divers Device/Protocol Controllers, discussed in the overview section, each have emulation components, shown in FIG. 4, that reside in either/both the VL and the PDI. The objective in any VCO emulation mode is to enable to the VDI to operate fully, without the ability of it, or any client applications, to differentiate between operation with actual device support, or emulation of it. There is an emulation mode flag in the system information that determines the operating mode. Any operations supported in the VL or the PDI must check this flag during every invocation of service, and either access a physical device, or provide a set of results

indistinguishable from having done so. No support for emulation of operations exists in the VDI -- it makes no direct references to low-level device services except through the PDI.

Remote Station Attachment Mechanism

Attachment of a remote station to the local station enables the local station to monitor its event stream and control its operations. The basic mechanism of attachment has been discussed in general, however the specifics of this interaction deserve more attention. Essentially, an attachment mechanism sufficient to monitor a remote station requires a cross-linking of the output of one station's event encoder to the other station's event decoder. To establish control of a remote station, there must, in addition, be a cross-linking of the output of one station's command encoder to the other station's command decoder. There are a number of operational considerations that become the subject of negotiation between the stations, once attachment is accomplished.

Monitor Context

Monitoring context refers to the station that is the source of the current event stream emanating from the local VCO's dispatcher. Any combination of the monitoring modes can be in effect once attached. A VCO that is not attached to a remote station can only monitor local events. The remote station must grant permission to be monitored by the local station by indicating so in its monitor parameters. VCO monitoring modes are described as follows:

- **Local** monitoring mode affects the target VCO to dispatch and process events local to it.

- **Remote** monitoring mode affects the target VCO to dispatch and process events emanating from the remote station to which it is currently attached.
- **Array** monitoring mode affects the target VCO to dispatch and process events from a specified array of remote stations.

Control Context

Control context refer to the station that is controlled by calls to local VCO SCI member functions. The remote station must grant permission to be controlled by the local station by indicating so in its control parameters. VCO control modes, from the perspective of the local station, and when set in the physical local station VCO, are described as follows:

- **Master** control mode enables the local station to control a remote station to which it is currently attached.
- **Slave** control mode enables the local station to be controlled by a remote station to which it is currently attached.
- **Peer** control mode enables both the local and remote stations (attached to each other) to control themselves, but not each other. This is the standard operating mode for most interconnections between stations.

CLIENT SOFTWARE ARCHITECTURE

FIG. 5 depicts a VCO client application; arrows highlight the flow of function calls through its various components. The diagram shows a full implementation of a VCO client application that best describes the VCO Client application concept. All of the components shown are not necessary to establish a minimally-functional multimedia

connectivity session with a remote station, but are needed to make full use of the entire VCO feature complement. The client application specification, unlike that for the VCO itself, is represented in a generalized fashion, and strict compliance is not necessary to achieve the benefits touted by the VMCS; a broad range of effective application designs may be derived from this prototypical VCO Client application model illustrated.

Summary of Client Software Architecture

A client application selects a class library supporting an implementation of class VCO. Constructing class VCO, it makes calls to the Software Control Interface for the newly invoked VCO to establish a connectivity session with a remote host. The client has a number of components that it uses to manage this session:

- *The Device-independent Connectivity Application Shell* provides the user-interface and basic task management for the client application. This component displays session status information, and initiates the milestones of its inception and termination. This component is the logical control mechanism for all VCO operations. If it is to construct a VCO directly, it must be created with the same object-oriented language as the VCO itself. While it is preferred that the client is a GUI application to best present the VCO control system to the user, it can be as simple as a daemon running in background, that processes string commands from a data stream directed to it.
- *A number of Notification Receiver Objects* receive notifications from the VCO that various VCO events have occurred. Client

applications typically create a Notifier Object to receive text streams from the VCO terminal output port. At least one other Notifier Object should be created to receive indication of the three major classes of new local station modes (new H.221 transmit modes), new remote station modes (new H.221 receive modes), and new call state changes (new call and line states) -- one Notifier Object can be configured to respond to all three of these event types.

- *Event and Text Processing Components* are specifically designed to analyze and/or respond to text and event information emanating from the VCO. Text processing of the terminal output stream can take the form of graphical display in a trace window that has the facility to enable its viewer to move forward and backward through the output messages, in order to view the progress of the session. Trace information could also be saved to a log file to permit later analysis of session activity. Finally, trace output can be analyzed by a debugger, or H.320 protocol analyzer. New remote station modes are usually routed to the Station Profile Controller for processing and interpretation.
- The *Station Profile Controller* examines new modes set by the remote station, and using a Station Profiler, compares them to a database, to determine the remote station's manufacturer or type. Once identified, the remote station's profile is elicited from that same database, and its non-standard feature set made available to the

application. Non-standard features include advanced camera control operations, proprietary VCO features, data exchange protocols, and application sharing features. Non-standard capabilities are also examined to determine the level of functionality, of which the remote station is capable. The Non-standard H.221 Mode Mapper provides a virtualized representation of the remote-station's available special features, and presents them to the application shell in a manner conducive to their mapping to local station physical controls.

Application Notification

In the general sense, a stream of events flows from the VCO to the client. Ongoing notification of the application, by the VCO, in the form of multiple concurrent event streams delivered to application class objects, changes the context of the VCO from a sub-system invoked by the client, that returns values in response to commands, to an adjunct connectivity operating system; an operating system running in parallel with the primary operating system, actively communicating with its client application processes through an interrupt-like mechanism, and similarly operating completely independently to specifically manage system multimedia connectivity resources.

Notifications from the VCO, to its client applications, take place using a mechanism designed to provide structured entry points that function much like interrupt service routines. From the perspective of the client, the design eliminates the risk of interfering with the delicate timing of the underlying multimedia connectivity sub-system, and does not confound normal

time slice allocations by the operating system scheduler. At the level of the application, notifications are discreet entities that are independent of any operating system or GUI event processing/queuing schemes, and resultantly more time-responsive; so much more responsive than adding events added to the application event queue, that notifications can preempt drawing operations by the GUI, but without inordinately starving the GUI of its time slices. The notification is usually implemented to run on a separate thread, concurrent with those drawing on the display, and thus connectivity events can be processed concurrent with drawing, rather than subsequent to a GUI operation in progress. The VCO Client notification system permits the design of high-performance, multithreaded applications that can process and respond to connectivity events with responsiveness that (in the context of a user interface) approximates real-time. Notification to VCO client applications, proceeds with rapidity such as is required for controlling both local peripheral devices, and the peripheral control features of remote stations, while concurrently maintaining a responsive graphical desktop display.

Notification Receiver Objects

In the application, any class object can be configured to receive calls from a Notifier Object when any one of a subset of events occurs. A member function in the object is declared for this purpose by the creation of a thunk. As previously described, thunks are created to redirect calls from the Notifier Object in the VCO, from a public global non-member function in the application (called by the Notifier Object), to the particular class object instance and member function intended exclusively for the purpose of notification. The

receipt of notifications by the application often results in the application's issuance of calls back to the VCO to correct, compensate, or respond to the condition. Many event handlers in the application function as feedback loops; upon notification, they immediately invoke VCO functions in response. Logical assistance by the client application is unnecessary once appropriate response routines have been setup - the VCO manages the multimedia connectivity system automatically.

Station Profiling

The primary objective of the client station profiling mechanism is to first identify the remote station as one represented in a local database of potential remote station types. Once a descriptor for the remote station is found in the database, the client can now determine any non-standard device modes (that invoke special features) supported by that remote station. Further, a list of corresponding non-standard capabilities is also stored in the descriptor, such that the local station can make a positive predetermination as to whether the special feature associated with the remote station type, is actually available for use. Nonstandard modes supported by the remote station can then be mapped to ~~local~~ controls so that the VCO client becomes capable of a degree of station control typically only possible by interconnection between two stations from the same manufacturer - VCO stations can fully understand the particular proprietary non-standard features they support. The VCO client is capable of an extraordinary degree of compatibility with an unlimited range of remote stations.

Station Profile Controller

This software controller contains three components that provide the functions necessary to implement a transparent mapping between local station controls, and remote station non-standard features. Local station features, beyond those represented in the VCO control mechanism have specific sequences of device modes that must be set to activate them. Non-standard modes on the remote station work the same way, except the mode sequences are different. The three components of the Station Profile Controller enable the client to associate any local or remote station non-standard feature (mode sequence) with a control on the local station. In short, the Station Profile Controller offers a symbolic, device-independent representation of local and remote station non-standard features, and beyond that, the ability to associate one local with one remote. An example of this association is in order: consider a surveillance system that maintains two specialized features:

1) It allows an operator to remotely select the current camera from a variety of available input cameras.

2) It displays an "X" cursor on the operator station video image, pinpointing the exact center of focus for its currently selected camera, and the remote station will move its selected camera to correspond to any mouse-invoked change in the cursor location on the operation display, therein allowing the remote operator to survey the area with simple mouse movements. The remote station will continuously reflect the camera's actual physical position by rendering a cursor on the operator station visual display. There is no H.320 representation of these

operations, beyond support for sharing a cursor position; the selection of a remote camera is a simple operation, but the second feature is one complex, proprietary, and in need of specialized library support features -- the cursor movement mechanism requires a complex feedback mechanism to move and display the "X" cursor as intended by the remote station's programming. When the operator station connects to this monitor station, the operator station determines that the remote station is of this particular monitoring station type, and locates a descriptor in its database that describes it. The modes to select the camera are represented symbolically to the VCO client, and mapped to local station controls. The sequence of mode-setting operations necessary to the selection of the remote station camera is invoked by offering the symbolic representation of the operation to the Station Profile Controller. For the more complex cursor-aiming feature, incoming cursor position modes are mapped to a virtualized definition of cursor movement, and passed to functions in a library of supporting routines, developed specifically to display it as required. Local station mouse movements over the video display region, on the operator station's bitmapped display, are mapped to cursor movements sent to the remote station via a similar mechanism used for camera selection. It is the VCO's notification mechanism that enables the concurrent processing of device modes,

sending and receiving them to/from the remote station, at the system level of the GUI application, without interference from other application activities.

Station Profiler

This component is responsible for identifying the remote station. Upon connection, it sets sequences of modes, and conducts whatever query is necessary to determine its manufacturer. To this end, it compares modes sent back from the remote station to stored sequences in the database.

Non-standard Mode Mapper

This component maps non-standard local modes, to specific features, by assigning mode sequences (and function calls) to an intermediate symbolic representation, which is then used in a feature mapping table. The same mapping is performed for non-standard remote station modes, however the mode sequences are preprogrammed, stored in a database descriptor, and selected according to the identity of the remote station.

Non-standard Capabilities Mapper

This component manages the capabilities associated with the non-standard modes handled by the non-standard mode mapper. It provides a mechanism to determine if non-standard modes are available on the remote station, as well as mechanism to inform the remote station that the local station is capable of handling its non-standard modes.

CLIENT VCO ACCESS METHODS

Derived from this design are several methods for VCO Clients to access the services of the VCO, so as to

make use of the VCO as an independent multimedia connectivity operating system that supports client sessions.

Notes On Drawing

FIG. 8 depicts the various methods used by VCO Clients to access service of a particular VCO. The left of the drawing, labeled "REMOTE SYSTEM", and the right, labeled "LOCAL SYSTEM", should not be confused with the "REMOTE STATION" or "LOCAL STATION"; access from another system may be from any computer supporting a text command stream to the host system, even from a dumb terminal. If the controlling system is a "STATION" (connected across the network), then it must establish a text data stream (in-band or out-band) to transceive VCO commands between the two systems. Note that the VCO depicted in the "REMOTE SYSTEM" is controlling the local system as its master, by employing some command dispatching mechanism to connect to the local station, but not necessarily over the network.

Summary of Access Methodologies

The services of VCOs are utilized by client applications that construct them, and subsequently make calls to their member functions. Once constructed, the VCO lies resident in the host system as an adjunct multimedia connectivity operating system that can respond to requests for service, when accessed in one or more of the following ways:

- Client applications running in the host system are able to construct one, or more, VCOs through the usual Direct Member Access method; that is, they call member functions in the VCO's Software Control Interface, to drive a multimedia connectivity session.

- To provide text command access from a remote system, without sending commands across a network connection, a VCO/TTY Access Daemon can construct a VCO in a host system, and then open a command text stream through a system communications port. Any remote system connecting to that port can send commands, and examine the effects of their issuance.
- A VCO terminal session is established upon connection to a remote system communications port that is being monitored by a VCO directly, or monitored by an Access Daemon. From the perspective of the remote system, the method of creating a terminal session to control a VCO, is referred to as Remote Command Access. Simply put, VCO commands are issued directly from a remote station or dumb terminal, to a waiting VCO, or daemon acting on its behalf.
- A seamlessly integrated remote VCO control solution, referred to as Remote Member Access, is an access method that creates, in a VCO implementation, a Media Control Object that is expressly designed to establish a bi-directional text data stream through a particular system port. The VCO command/event streams are directed through it, to provide a level of control that allows a VCO client, invoking VCO members on its own local system, to drive a remote VCO transparently. This method utilizes the identical components and mechanisms as for remote VCO control across a connection, except that the command/event stream is directed to an out-band

communications port, and not the principle network connection.

IMPLEMENTATION

This section describes the full implementation of a VMCS that supports concurrent live audio, live video, imaging, and binary data transfer services. The VCO portion of the system must be created to support the specific configuration of devices installed. Compliant client applications will run over any VCO that they construct. Any number of VCOs can be created to encapsulate divers combinations of devices installed in the system. An instance of a VCO (that encapsulates a device set) is one of many possible presentations of that same device set to an application; a different VCO implementation may invoke the same devices in a different way, or using different drivers (for example) to present an entirely different performance profile. Depending on the capabilities of the sub-systems installed, multiple VCOs can be instantiated concurrently to provide multiple multimedia connectivity sessions at the same time. There is no limit to the application of the VMCS paradigm, as long as the specified VCO service is provided, through some means, to the client application or marked as absent in the VCO's capabilities listing.

VMCS HOST SYSTEM REQUIREMENTS

Implementation of a VMCS is best accomplished in a microcomputer host system equipped with peripheral devices to process audio and video signals, and a connection device to interface with the network. A VMCS can be constructed to run over almost any combination of the components discussed in the section entitled Host System Equipment Requirements below, depending on the level of implementation desired; support for concurrent

audio, video, image, and data services is hereupon described for the disclosure of full VMCS implementation, but any partial implementation is possible without affecting the basic VMCS design. The VMCS is ideal for limited usage i.e. only for audio connectivity. Furthermore, the bewildering array of devices for audio, video, data, imaging, and videoconferencing, that are now available, often combine the functionality of two or more devices, in which case the perceived differentiation between them exists only in the software abstraction layers that comprise a VCO. For example, an audio and video device may be combined on one board, but will appear as (map to) a number of discreet functional Media Control Object entities, whose hardware support mechanism is indeterminate, when considered at the level of the VCO Client application.

Host System Equipment Requirements

Following are the requirements for the host computer, adapter cards, peripherals, and system software components requisite to VMCS implementation, as already described. Each item is intended to represent an example component; many permutations of features and hardware configurations are acceptable for actual deployment, though the configuration outlined below is provided in specific terms to enhance the clarity of subsequent references.

- **IBM-compatible Personal Computer**
  A Personal Computer is the preferred host system. It should have a Pentium processor running at 120Mhz or faster, contain at least sixteen (16) megabytes of random access memory, and a minimum of 500 megabytes of backing store capacity.

- **32-Bit Multitasking Operating System**
  The VMCS host operating system should provide protected memory address space for each process, support multiple threads, and have a preemptive scheduler.
- **Graphical User Interface**
  A VMCS host operating system user interface should be event-driven, and provide a windowed graphical "object-desktop" environment where each visual component can be manipulated by drop/drag/cut/paste/properties operations.
- **Audio and Video CODEC Devices**
  Audiovisual encoding and decoding hardware may be integrated with other devices onto one or more adapter boards that plug into expansion slots in the computer. The CODEC devices for this implementation must encode audio and video inputs from a microphone and camera, respectively, to a multiplexed digital signal compliant with the H.221 frame structures. Decoding must proceed from this H.221 compatible signal to an analog audio output, and a VGA video signal for output to (for example) a video display terminal.
- **Video Display Adapter with Overlay Controller**

  A high-resolution video graphics adapter must be installed so as to work in conjunction with a video overlay device. This hardware configuration will support the station's principle visual graphics information output pathway by enabling the simultaneous display of bitmapped graphical and motion-video. This sub-system must permit motion-video display

in a windowed portion of the main screen, a region programmatically selected for that purpose. The overlay controller allows the display of motion-video over a region of the bitmapped display device by enabling the real-time overlay of NTSC video frames onto the identified region of the main display bitmap.

- **Audio and Visual Peripheral Transducers** These peripherals include input devices such as an NTSC camera to input motion-video, a microphone to input audio, and a 600 DPI color scanner to input high-resolution still images. Output devices include a 17" CRT display (1280 x 1024 resolution that can display 65,535 colors) for bitmapped and motion-video output, a loudspeaker for audio output, and a color laser printer for hardcopy photograph and document output. Audiovisual devices may plug into analog signal ports on adapter boards designed specifically for the purpose of PC bus interface, or into standard digital computer ports (according to their own unique interfacing requirements).
- **Media Access Control Device Drivers.** Device drivers must be provided for the audio and video adapter boards (including the overlay controller) enabling the initialization, configuration, shutdown, and querying of each. System device drivers must be available to input scanner images, output printer images, and control system data ports, among other standard system services.
- **Network Interface Unit (NIU)**

A network interface unit must provide the physical link to the network. It needs to support a minimum transfer rate of 128 kbit/s through a plesiochronous network (see U.Black, ATM, Foundation for Broadband Networks, Englewood Cliffs, New Jersey, Prentice Hall PTR, pp. 36-37, 1995) such as that provided by the Integrated Services Digital Network (ISDN). In the case of a host PC, the physical network connection extends to an ISDN phone jack, from an adapter card plugged into one of the computer's expansion slots. A fully-digital ISDN modem device is usable for this purpose.

- **Network Protocol Stack**

The network interface must provide programmable software control of the physical network service, and in the case of the recommended ISDN configuration, ISDN network protocol software must provide accessibility to one or more b-channels (for encoded audio/video data) and to a d-channel for the out-band signaling required for H.320 protocol implementation. Data packets from the system must be directly accessible for synchronous transfer to and from the decoder/encoder devices (audio and video CODECs).

- **ISDN Basic Rate Interface (BRI)**

For the preferred ISDN network interface, the ability to establish a connection supporting 128 kbit/sec is generally accepted as the absolute minimum bandwidth needed to support a primitive motion-video image (with a concurrent audio signal) across the ISDN. A

typical BRI installation is utilized as a 2b+1d channel configuration. For most purposes, a triple-BRI, or composite line configuration (384 kbit/sec) is preferable, as it is capable of producing an image closer in quality to that is generally considered acceptable in other video applications.

System Development Requirements

Developing a VMCS from preexisting hardware components is a combined system and application software development effort. Initial development of a VCO to control a set of devices is a significant undertaking that involves careful interface to device control software, and implementation of many of the specific protocols residing under the H.320 rubric. While implementation of the prototypical VCO kernel is non-trivial, diligence is repaid many times over in that nearly all of the kernel source code is propagated, in rote fashion, to create a new VCO that can readily support a new set of devices. The client application binaries are directly re-releasable -- client programs are fully device-independent and run over any VCO built to specification. Requirements to implement a VCO are outlined as follows:

- **VMCS Disclosure** provides the necessary description of a Virtualized Multimedia Connection System to derive a design for an actual implementation. A complete set of the ITU-T recommendations referenced in ITU-T Recommendation H.320, are a necessary adjunct to the implementation and testing of fully compliant protocol implementations (see REFERENCES).

- **C++ Software Development System** or a functionally equivalent object-oriented development system must be used to create both the VCO (server) and client portions of the VMCS. Full implementation of the referenced AT&T C++ language must be supported.
- **Developer Toolkits** provided by hardware and software OEM's, whose components are to be incorporated as VCO components, is essential to porting the device-independent VCO kernel to a new multimedia connectivity sub-system. Software tools to create the graphical user interface modules (such as exception handler message boxes and configuration displays) must also be available.

Software Development Considerations

To restate the purpose of VMCS server components: it is the primary directive of every VCO to bind, dynamically at run-time, a connectivity source to a set of transducers; and do so in such a way that the service provided to client applications serves as a mechanism to share spectral information between interconnected stations. In the use of it, no consideration is given to any intermediate data transmission methods employed. It is the responsibility of the VCO implementer, to ensure that sound and light directed to and from the remote station, are somehow seamlessly, automatically, and transparently transited over the void that may exist between data streams associated with the source of connectivity, and those associated with the local transducers. Most systems utilize an integrated audio/video hardware design to provide a direct analog signal link between these parts -- consider those

manufacturers mentioned in the Background section -- but this model is crippling to the station in its other purposes for reasons aforesaid.

The operating system type specified for this system is characterized by the ability to spawn threads that run concurrently at specified priorities. They can be utilized to support transparent (to applications running in the foreground) real-time data streaming facilities. Data streams to/from connectivity sources can be attached to/from transducers, so as to bridge any gap between discreet devices installed separately in the system; sharing data between separately installed devices requires read/write operations executed by the microprocessor (there is no direct analog signal connection between devices on different adapters). With the specified operating system type, a station can take advantage of the multiprocessor personal computers that would support the transfer of data at very high speeds (between devices in the system), even at rates sufficient for normal system operation, while processing audio and video in real-time.

Whatever mechanism is used, hardware or software, the VCO implementation must create the operational context in the system, dynamically when invoked, to move data between system components. It must do so in a way that is fully protected, secure, and unaffected by other system activities. The creation of the sub-system's operational context must be transparent to the client application, as must its destruction at session's end.

VCO SOFTWARE IMPLEMENTATION

A VCO is mostly comprised of software objects whose member function implementations are overridden by more derived versions of themselves that provide structured access to the services of installed adapter

devices. As the VCO architecture is a direct application of software object technology, to elucidate the details of its embodiment entails discussion of its components in terms of a class derivation hierarchy. Next follows examination of the VCO's class structure.

VCO Class Derivation

One VCO implementation encapsulates one specific sub-system configuration that exhibits a particular set of properties (capabilities) that defines its unique service profile -- unique in the set of standardized VCO operations it can support, but no different from any other VCO in the way it presents them to client applications. Correspondingly, each VCO is no different from any other in the way it is implemented. In fact, there are a number of implementation principles to consider prior to VCO design specification, thus speaking generally towards application of the concept...

- *More derived* classes in the VCO are more device-dependent, ranging from the device-independent VDI classes, to the device-dependent class PDI.
- *Less derived* classes in the VCO are less device-dependent, wherefore all VCOs contain the same device-independent kernel (that is comprised of class VDI and all those from which it is derived).
- *More derived* classes are more time critical, ranging from the VDI that responds to occurrences of events in OS-scheduled time slices, to the PDI that can queue events during interrupt service routines, invoked in real-time by device requests for interrupt.
- *More derived* implementations (more device-independent default implementations) of VCO

virtual members substituted with a more suitable implementation overriding them with one more device-specific (residing in a more derived class); that is, a default function is provided by the VDI, which the programmer can override in his particular implementation of the same feature.

- In most cases, *more that 90%* of the VCO source code is reused in the next VCO implementation without modification, due to the imposition of rigorous functional constraints (by the VCO architecture) on its class structure.
- A pure virtual member interface in the device-independent VDI, to more derived device control members in the device-dependent PDI, impose strict isolation of logical from physical operations. This isolation of logical operations from physical device operations, is realized by exploiting object-oriented software language constructs integral to the language itself; structural integrity and layering of operations in the VCO is enforced at the most fundamental level of source code expression. The device control members used by the VDI (to lend physical device control to the implementation of its algorithms) are accessible directly in the same class, but the underlying device control mechanism is (for all intents and purposes), in one more derived and not directly addressable. Resultantly, any changes to the way these more derived device control members are implemented, are beyond its discerning.

- Design-level isolation of logical and physical device control mechanisms in the VCO architecture, are incorporated so as to intentionally expose a well-defined, readily exploitable "fissure" in its layering model, whereby the core technology is rendered amenable to specific extensions of concept. The implementations of certain system designs are significantly reduced in expense and difficulty as a direct result of the well-defined logical-to-physical mounting mechanism. Some applications of concept advanced by its accessible mounting mechanism include:
  - Rapid prototyping and redeployment for new sub-system configurations
  - Distributed VCO development by VDI and PDI development teams
  - Microcoded or embedded PDI implementations
  - Distributed media control systems
  - Remote station control and diagnostics

The class derivation diagram depicted in FIG. 6 shows the classes that directly comprise the VCO, as well as adjunct classes (Notifier and MCO) that are used to implement its feature set. Every component shown is used in every VCO implementation exactly as shown. Class VDI, being the Virtual Device Interface used by clients to access the VCO's encapsulated sub-system, is the only class, besides the public constructor and destructor in class VCO, that contains public member functions. The symbols used to describe the various relationships between classes are mostly proprietary to this disclosure, as no widely-used convention to graphically

express object-technology concepts has been adopted by a major standards organization. Symbolic conventions used here are shown in FIG. 1.

Summary of VCO Class Derivation

Each VCO is a composite derivation of the six classes: Terminal, Exception, Event, VDI, VL, PDI, and VCO. In the order of least to most derived, the derivation sequence for the VCO itself, proceeds as follows:

- **Class Terminal** enables the VCO to send text messages to a set of character output devices, or receive text messages, that are subsequently interpreted as commands. Since the VCO terminal can be programmed to use the VCO notification mechanism as a virtual output device, the class contains a pure virtual member used to direct text output to a Notifier Object configured for such purposes. This member is overridden by a supporting member function in the more derived class Event, and this override must be present for class terminal to compile.
- **Class Exception** is derived from class Terminal, and is defined to contain member functions and data related to reporting fatal errors by responding in some pre-configured way. In the most primitive sense, the only service that class Exception must be able to access is some method to relay the fact that the exception occurred, and by inheriting members of class terminal, it can. Class Exception also has a virtual function used to shutdown the VCO when an exception occurs. An override in the VDI provides an

implementation that shuts down the VCO, as expected during fatal errors.

- **Class Event** is derived from class Exception, and is defined to contain the VCO event manager, which in turn manages the notification mechanism. It maintains a linked object list of Notifier Objects which are themselves each individually derived from the NOTIFIER data structure. Every Notifier Object is a protected class that is created by class Event, and is a friend to it. Class Event, but no other, can create, delete, or access class Notifier members directly except members of class Event, thus the Notifier Objects are essentially creatures of it. The event handler is the VCO's "proxy" to the linked list of Notifier Objects. Class VDI is a protected derivation of class Event. It is defined to contain a large set of members that comprise the VCO Software Control Interface, and a number of device-independent protocol support procedures. Inheriting the services of classes Terminal, Exception, and Event, it is capable of presenting the entire VMCS connectivity services through the member functions of a single binary software object; that is, one instantiated upon construction of a class derived from it.
- **Class VL** is derived from class VDI, and provides a location for an implementation-dependent set of routines to map physical device control operations and responses to/from the logical representation manipulated by members in the VDI. Most of its members are private to it, and narrowly

focused in scope. Entry points to translation and mapping services are made public to more derived classes that wish to utilize them.

Class PDI is derived from VL, and contains within it, private definitions and implementations of member functions that override the pure virtual device control members used in the VDI to invoke the services of physical devices in the encapsulated multimedia connectivity sub-system. The implementation of the pure virtual overrides utilize members in the VL to translate and map the structure of arguments and input/output data syntax to that expected by the VDI implementations. The PDI contains mechanisms to access the VCO Media Control Object Device Control Mechanism (Media Control Objects). This mechanism relies upon the maintenance of a linked object list of instances of class MCO maintained much like the linked list of Notifier Objects in class Event. Class MCO is derived from an MCOPARAM data structure that serves as a general purpose repository for device control information as associated to a particular signal from a particular device. As with the administration of Notifier Objects, instances of class MCO are directly accessible only by the PDI; only the PDI may directly examine, modify, and invoke their members. The design of the VMCS is to promote its utilization in a variety of operating environments that include distributed systems, remote access across a network connection, and text command (teletype)

interface via dumb terminal. Members of class MCO are accessible to underlying Media Access Control software, and MCO implementations make calls to the same to invoke device services.

- **Class VCO** is derived from class PDI, and functions as the capstone for the VCO class structure. The only members it inherits from its parent classes are the public members that comprise the SCI in the VDI. Its constructor and destructor call those of its parent classes, thus it invokes those of the VDI to create and destroy the VCO session. Class VCO contains all additional implementation-specific entry points (object members) that are presented to client applications, including extensions to the VMCS that are not directly related to controlling the connectivity session. All client applications proceed with the expectation that the invocation of VCO services will always be accomplished using a pointer or reference to class "VCO".

Class Components

Next follows detailed descriptions of the operations that must be implemented by each class comprising the VCO.

Class Terminal

Provides full implementation of the VCO terminal services by maintaining a list of output devices for the output terminal, and writing all text message sent to the terminal output port to every device on this list. Text messages sent to the terminal input port are assumed to

be VCO text commands. They are parsed into tokens, decoded, and executed as SCI commands. The sub-components residing in this class are listed below, with an accounting of the specific operations for which they must provide support.

- Terminal Stream I/O Device Controller
  Operations supported by this sub-component must include the following:
  - Add output device to list
  - Remove output device from list
  - Write text message to output device
  - Text Command Decoder

  Operations supported by this sub-component must include the following:
  - Parse text command message to command token list
  - Verify command syntax
  - Execute command token list as SCI command
- Text Command Encoder
  Operations supported by this sub-component must include the following:
  - Verify command syntax
  - Translate SCI call to text command message

- Linked Terminal Stream I/O Device List
  The list itself is implemented as a linked object list, where each object contains the member functions and member data necessary to transmit data to the file, device, signal, or data port referenced by it.

Class Exception

Provides full implementation of the VCO exception handling operations that include reporting the occurrence of the exception, and subsequently shutting the VCO down. Additional features of this component include the maintenance of an enable/disable flag that is tested by every public member function upon entry into the VCO; a disabled VCO must reject any call into it, and return the "Disabled" result code to the caller. There are a number of flags that can be used to configure the exact modality used by the exception handler to respond to exceptions, and each modality must be supported by the exception handler implementation, in accordance with the definitions shown in FIG. 6A. The sub-components residing in this class are listed below, with an accounting of the specific operations for which they must provide support.

- Exception Handler
  Operations supported by this sub-component must include the following:
  - Process VCO exceptions accordingly (see FIG. 19)
  - Provide for capability to display debug information message box
  - Display "user" information message box
  - Send text information message to terminal
  - Trigger Notifier on exception
  - Trigger VCO disabler mechanism on exception
- Disabler Mechanism
  Operations supported by this sub-component must include the following:
  - Maintain VCO enabled/disabled flag
  - Provide for query of enabled/disabled flag (see FIG. 19)

- Invoke system shutdown utilizing virtual override in VDI

Class Event

Provides full implementation of the VCO event management operations. A list of Notifier Objects is maintained, and a mechanism to trigger them is contained in this sub-component. The VCO event queuing and dispatching mechanisms are located in this component, though critical section handling may be located in the VL to make use of special operating system support for semaphores and thread blocking features. There are thirty-two (32) distinct events that have been standardized for VMCS use. FIG. 6C shows the symbolic identifiers for these events, and provides concise definitions for each. The physical source of the event is labeled as either hardware (HW) or software (SW), and accompanied by a code that goes on to further clarify the specific system component from which the event is likely to (though not necessarily) emanate.

VCO developers creating a VCO to work with a new device set, must identify the most reliable source for VCO events originating in hardware, and then map vendor-specific representations of the event to those virtual, standardized, and described in FIG. 6B. Third-party device drivers in the MAC layer may not provide access to events identical in meaning or context to those cataloged by the VCO; some interpolated or emulated derivation (from events closely related) may be necessary for a compliant indication of the standardized occurrence, and any member functions created to approximate the representation of one such only marginally identifiable, should reside in the VL layer for invocation by members in the PDI.

The event manager is also responsible for managing the flow of trace information to its terminal output

port. The sources from which trace information emanates, can be programmed in an additive fashion, by specifying a trace output profile. There are a number of flags, applied to express this profile as a logical combination of trace output source locations within the VCO's works, and each modality must be supported by the event manager implementation, in accordance with the definitions shown in FIG. 6B. The sub-components residing in this class are listed below, with an accounting of the specific operations for which they must provide support.

- Notifier Object List Manager
  Operations supported by this sub-component must include the following:
  - Create and add new Notifier Object to linked Notifier Object List
  - Remove Notifier Object from list and delete
  - Set Notifier Object triggers
  - Get Notifier Object data
  - Lock Notifier Object List against add/remove operations while triggering
  - Unlock Notifier Object List to allow add/remove operations
- Notifier Object List
  The list itself is implemented as a linked object list, where each object contains the member functions and member data necessary to configure the Notifier Object's triggering profile, as well as to actually trigger it to deliver notification to its associated Notification Receiver Object.
- Notification Triggering Mechanism
  Operations supported by this sub-component must include the following:

- Trigger Notifier Objects in Notifier Object List (see FIG. 10)

- Event Dispatcher

  Operations supported by this sub-component must include the following:

  - Dispatch event (see FIG. 11)
  - Start dispatcher
  - Stop dispatcher
  - Set dispatcher rate
  - Configure dispatcher

- Event Queue

  Operations supported by this sub-component must include the following:

  - Add event to queue (see FIG. 11)
  - Remote event from queue (see FIG. 11)
  - Flush queue

Class Notifier

Provides full implementation of the VCO Notifier Object (NO). Each NO is a self-contained reporting mechanism called a thunk. This thunk must be created by any class that wishes to be informed of the occurrence of a VCO event. The thunk provides a globally defined entry point to the address space of the instantiated class object that is to receive notifications, and the thunk retains knowledge of the exact class name and specific class member designated to receive notifications (from the NO residing in the VCO itself). The NO stores a pointer to this global entry point (the thunk), a pointer to the Notification Receiver Object (NRO), and a pointer to the NRO's Notification Receiver Member (NRM) that is the ultimate destination for delivery of notification. The NOTIFIER data structure from which the Notifier Object is derived, contains all of this information, the achieved objective in its tracking to enable an immediate conveyance of a unit of system event information (a

standard VCO event) directly from a driver-level component to the application, as soon as there exists an opportunity for the operating system to run the interested application. With regards to VMCS implementations and their notification mechanism, system designers should first reflect upon the following:

- Notifications are designed specifically to operate like system interrupts, independent of user interface event queues. Like interrupts, they require service only in response to very specific occurrences to which they are programmed to respond -- service routines do not have to test for a wide range of possible triggering events, but can act directly with simple, well-defined operations.
- Notifications from the VCO are virtually unaffected by user-interface operations, and events are never lost to "queue-full" conditions. They are fast, configurable, flexible, and offer a measurably more reliable feedback mechanism than the typical GUI event delivery mechanisms, but expectantly can interrupt drawing operations in progress. Drawing operations, to display information delivered by a Notifier Object, are best executed from a specific painting routine, whose invocation is governed by the receipt of paint messages from the GUI -- painting messages and graphics to the display with each notification can prevent the GUI from processing messages in its own event queue.
- Consistent with the previous point, NRMs should be constructed as high-level interrupt

service routines that insert an event into the application's event queue (GUI event stream), or spawn a new thread to exact some effect on the system, and return immediately. To delay processing on a notification thread could delay notification to other VMCS objects (if a single thread is used by the triggering mechanism to trigger all NOs). Beyond delay, none of the usual problems associated with delayed interrupt processing occur since the VCO queue retains all events till processing is resumed; no information is ever lost. Further, VCO events are but shadows of real-time events that will have long since been serviced in real-time, according to the methods implemented by driver-level vendor-specific components. However, correct designs for systems will service all outstanding event notifications and return long before the VCO dispatcher is ready to remove another event from its queue.

The constructors of Notifier Objects add them to a linked object list upon construction, and remove them from the list upon destruction; their structured integration into a linked storage format is managed automatically. An accounting of the specific operations for which this class must provide support are listed below:

- Notifier Object Operations

  Operations supported by this sub-component must include the following:

  - Find Notifier Objects to trigger in linked object list (see FIG. 10)
  - Trigger individual Notifier Object in list (see FIG. 10)

- Set Triggers
- Get Statistics
- Reset Statistics

Class VDI

Provides full implementation of the VCO Software Control Interface (SCI), along with a number of private support functions. Any device-independent routine necessary to VCO implementation resides in this class. The header file VDI.H (see Appendix) contains all of the constants, enumerations, and data structures used by both server and client portions of the VMCS. Following those definitions, is that of the SCI. These member functions are the virtualized definition of the VMCS control mechanism from the client application's perspective, and are the only public members of the VCO; notably excepted is the VCO constructor and destructor in class VCO. Not shown in VDI.H are the device-independent call and protocol management routines that provide support for the VCO connectivity sessions. They are implementations of various Recommendation H.320 protocols. The session-related sub-components residing in this class are listed below, with an accounting of the specific operations for which they must provide support.

- Network Session Operations

  Operations supported in this group of member functions have public entry points that are represented in the SCI (see Section entitled VDI Header File in Appendix). They are noted here to reference figures that detail their flow control pathways. The Network Session operations must include the following:

  - Construct VCO (see FIG. 24)
  - Destruct VCO (see FIG. 24)
  - Open VCO (see FIG. 25)

- Close VCO (see FIG. 26)
- Make call to remote station (see FIG. 27)
- Execute multipoint call operation (see FIG. 28)
- Hang-up call or line (see FIG. 29)

- Media Control Object Device Control Operations

Operations supported in this group of member functions are accessed by the public audiovisual/data signal switching control members in the SCI (see section entitled VDI Header File in Appendix). A query flag passed during a call to this member function determines whether or not a request for service or capability query is invoked. Operations supporting the service and query requests must include the following:

- Media control operation service request (see FIG. 20)
- Media control operation query request (see FIG. 21)

- Device-independent Call Controller

Operations supported in this group of member functions respond to line state events from the Device Event Processor, to manage the connectivity session. Call control related members must include the following:

- Enter call controller and process line event (see FIG. 14)
- Execute procedure to handle incoming line disconnected (see FIG. 15)
- Execute procedure to handle incoming line dialed (see FIG. 16)

- Execute procedure to handle incoming line ring (see FIG. 17)
- Execute procedure to handle incoming line ringback (see FIG. 16)
- Execute procedure to handle incoming line connected (see FIG. 16)
- Execute procedure to handle incoming line busy (see FIG. 16)
- Reset call data to default states (see FIG. 18)
- Restore default connected device modes (see FIG. 18)
- Set connected device modes (see FIG. 18)

- Device-independent Capability Exchanger Operations supported in this group of member functions contribute to implementing an algorithm that employs an H.221 mode-capability cross-reference table to determine if the connection between logical and remote stations can support a given H.221 device mode; that is, it compares the capability associated with the mode to the logical intersection of the capabilities of the local and remote stations. Capability exchange operations are internal to the VCO, and must include the following:
  - Accessibility to an H.221 mode-capability cross-reference table
  - Determine if connection supports mode (see FIG. 12)
  - Determine if capability is associated with mode (see FIG. 13)
  - Determine if local station supports capability (see FIG. 13)

- Determine if remote station supports capability (see FIG. 13)

- Device-independent Device Event Processor
A member function in the VDI is a Notification Receiver Member that receives notifications of device events from a Notifier Object created by the VCO at the time of its construction. Any events in FIG. 6C, whose source is identified as hardware, is considered a device event, and the Device Event Processor responds to them to maintain the current connectivity session. Device events are processed according to a specific algorithm that routes them through appropriate control flows depending on their particular category (see FIG. 23).

- Pure Virtual Device Control Members
There are no implementations of these members in class VDI; only the member declarations are present (see section entitled VDI Header File in Appendix). These members are used extensively by implementations of other VDI members to provide the device interface for all operations that access vendor-specific driver components and their underlying physical devices.

Class VL

Provides full implementation of any members necessary to convert, translate, map, or interpret operations and data formats between conventions used by logical controls in the VDI, and Physical Device Interfaces accessed by the PDI (MAC layer components). This class may vary greatly in the number of member functions it must contain for a particular VCO

implementation. The header file PDI.H (see section entitled Physical Device Interface File in Appendix) contains a definition of an empty class VL to show its role in the derivation of the VCO. Virtualization operations are unlikely to be device-independent, however the categories of operations they commonly must implement are preserved across most VCO implementations, and are as follows:

- Virtualization Operations
  - Software Component Load/Initialization
  - Software Component Unload/Shutdown
  - Configuration Information Access
  - Data Exchange Syntax Mapping/Emulation
  - Call Event Mapping/Emulation
  - System Information Mapping/Emulation
  - Capability Exchange Mapping/Emulation
  - System Exception Mapping/Emulation
  - Media Access Control Mapping/Emulation
  - Protocol Mode Mapping/Emulation

Class PDI

Provides full implementation of the VCO device control interface, including a number of operations to interface operating system services. A pure virtual override member must be implemented to support the operations defined for the pure virtual device control members defined in the VDI (see section entitled VDI Header File in Appendix). The header file PDI.H contains the definition of class PDI (see Appendix) and shows its role in the derivation of the VCO. Implementations in class PDI have no restrictions on the way they interface devices. Media Control Objects (instances of class MCO), are the preferred mechanism. At the time the VCO is opened, when its devices are initialized, the PDI creates an array of Media Control Objects that describes the

available, and expected-to-be-available, audiovisual/data signal types. These Media Control Objects contain the member functions and data structures needed to control the device that is the source of, or destination for, the signal they represent to the VCO. The next section entitled MCO Device Control Mechanism goes on with a further discussion of the Media Control Object Device Control Mechanism. The sub-components residing in this class are listed below, with an accounting of the operations for which they must provide support.

- Pure Virtual Device Control Member Override Operations
  Operations supported by this group of members are best described by the descriptions of the pure virtual member functions defined in class VDI (see "Class VDI"). The pure virtual overrides residing in class PDI are implementations of the pure virtual device control members declared in class VDI (see section entitled VDI Header FIle in Appendix).
- Device Capability List (H.221 Capability BAS Codes)
  A list of device capabilities is stored in the VDI (see section entitled Bit-Rate Allocation Signal Header File in Appendix), but must be maintained by the PDI. A local capability list in the VL is copied into the VCOPARAM structure in the VDI during VCO construction. Incoming capabilities from the remote station are written to the remote capabilities field of the VCOPARAM structure, by callback members in class PDI, and are called by the connectivity protocol stack when capabilities are transmitted from the

remote station. VCO device capabilities are represented as device-independent constants, so there may be a necessary mapping operation from the BAS code (or proprietary) representations used by the connectivity protocol stack to/from those defined for the VMCS.

- Device Modes List (H.221 Device Mode BAS Codes)

A list of all H.221 device modes is kept in the PDI as a reference. This list is used to determine if a mode is standard, non-standard, of a given type, or invalid.

- Device Event Linkages to Queue

In order for the PDI to be informed that device events have occurred, a number of callback functions are declared in this class. Such callbacks can typically be classified and implemented as follows:

- Connectivity Protocol Stack Callback Members are called by routines in the software modules that implement the connectivity protocol. In connectivity protocol stacks encapsulated by the VCO, the callbacks come from the OSI transport layer (or its equivalent). They call the VCO to report any changes in line states, the arrival of BAS codes from the remote station, and for a wide variety of status-related events, as defined in FIG. 6C.
- Media Access Control Callback Members are called by routines in the device drivers that comprise the MAC layer. They call the VCO to report changes in

device states, results of device operations, and a wide variety of status-related events, as defined in FIG. 6C.

Upon receiving an event from any callback function, the vendor-specific event is mapped or translated to one of the standard VCO events, and added to the VCO event queue. Routines in class VL may be called for this purpose.

Class VCO

There are no specific components to implement in this class. Extensions to the VCO feature set, beyond those related to control of the encapsulated sub-system, should be added as members to this class; it is the place specifically intended for such enhancements. The header file VCO.H contains the definition of class VCO (see section entitled General VMCS Header File in Appendix) and shows its role in the derivation of the VCO. All client applications must include this file in order to access the services of the VCO itself. Class VCO is the class that is constructed by the client, and it presents to this client a number of public member functions described as follows:

- Public VCO Members Available to Client
  - VCO is the constructor of the VCO, and invokes the constructors of all less derived classes when invoked.
  - ~VCO is the destructor of the VCO, and invokes the destructors of all less derived classes when invoked.
  - Inherited Public Members from the SCI are all presented to the client application as members of class VCO.

- Implementation-dependent Extensions can declare public member functions in class VCO that offer their services to the client application seamlessly with other VCO functions.

MCO DEVICE CONTROL MECHANISM

The device control diagram depicted in FIG. 7 shows how the VCO is able to reference the devices in its encapsulated sub-system as configurable representations of the data streams that they generate, process, or redirect. Client calls to the SCI are shown to invoke SCI members that, according to their specified function, rely on calls to pure virtual device control members for their implementation, thus not all SCI members are included in the diagram. The sixteen default Media Control Objects are arranged in the drawing to clearly demark the low-level, vendor-specific component to which they correspond, and manipulate when affected by PDI calls to their members. Vendor-specific MAC components should be considered bi-directional -- they support control pathways and data streams to and from a media control sub-system -- and the different types of transducers required for each direction are clearly evident. Concordantly one finds the "audio" objects reference a MAC component supporting audio input and output, and to that single MAC component is connected both microphone and speaker. PDI calls made directly to the connectivity protocol stack and MAC components are not shown explicitly in the drawing, as the exact structure of their interactions are left to the implementer's discretion.

Summary of Device Control Mechanism

Client calls to the SCI invoke members that often require the support of the encapsulated multimedia connectivity sub-system in their implementations. The implementations of SCI members, such as "Open" and "Call", make requests for physical device control services by utilizing at least one of the pure virtual device control member functions declared in class VDI; these members are private and entirely separate from the public SCI members offered to clients. The PDI contains overrides for these pure virtual device control members. These overrides invoke the appropriate device operations by making calls directly to the connectivity protocol stack or MAC layer components, as is appropriate to the desired device control operation. Implementations of the pure virtual device control overrides must perform any and all interface to vendor-specific hardware and software components necessary to fulfill the specified expectation of the pure virtual device control member in the SCI.

If the particular SCI member, called by a client, is MediaControl, the method of interface to the physical device is different from that used for call and protocol operations; its purpose is to switch or configure the audio, video, image, or data signals represented as virtualized system objects, or Media Control Objects. In this case, the pure virtual override for MediaControl, implemented in the PDI, then manipulates the members of Media Control Objects that have been created to represent the various available signal types. Depending upon the exact nature of the request, audio, video, image, and data signals are combined, redirected, displayed, or routed to local devices or the remote station. The Media Control Objects can also be used to set various modes (associated with the signals) by directly controlling the

device associated with it. The operation of the Media Control Object device control system is detailed as follows:

- Primarily there exist sixteen default Media Control Objects, as shown on the right side of FIG. 7.
- There is an input and output object to/from the remote station for each signal type (see FIG. 7A), for a total of eight objects representing information shared between the local station and the remote station.
- There is an input and output object to/from a local device for each signal type (see FIG. 7A), for a total of eight objects representing information shared between the local station and its local environment.
- The objects are only created when the signal they represent is within the capabilities of the system to support. They are only enabled when the signal they represent is actually available for access.
- Any number of Media Control Objects can be created in the VCO to control more devices and data channels, as determined by their detected system device availability by the VCO.
- Each Media Control Object, representing a specific signal type and direction, can be attached, or "plugged into" another Media Control Object that is compatible in both signal type and direction; For example, an audio source from a local device (microphone) can be attached to an audio output to a remote station, or a video input from a

remote station can be attached to an output to a local device (video display)

- Each Media Control Object, regardless of signal type, contains a data structure that reflects the various states and modalities of the signal. Member functions for each Media Control Object, allow them to be manipulated as independent, uni-directional channels.
- Certain Media Control Objects can be combined into composite media control objects that describe a complex signal type, such as multiplexed audio/video information. The objects can also be combined with objects that subject them to a specific transform, or split/join configuration.
- Setting the parameters of a source/input object invokes a sub-system (driver-level) attempt to change the settings of the source device or station, whereas setting the parameters of a destination/output object attempts to change the settings of the destination device or station.

Audiovisual/Data Signal Switching Mechanism

Device control operations are made directly available to VCO Client applications through the implementation of the MediaControl SCI member function, along with some related members that assist in the manipulations of these objects. The SCI members map to essentially identical pure virtual device control members implemented in the PDI. The switching of signals and device modalities generally takes place by selecting constants from various enumerated categories (see section entitled VDI Header File in Appendix), and presenting them to the VCO with the MediaControl member. The format

of the arguments is constructed so that the specified operation applies to the currently assigned default Media Control Object for the specified Media Control Object type (see FIG. 7A). For example, a command to mute the input microphone would likely reference AudioSrc as the Media Control Object type. Handles are used to assign various non-default Media Control Objects as the default (one of the sixteen) for a given type. The continuous linear enumeration of all possible constant arguments used for MediaControl function calls give each setting a unique numerical identifier, and thus each can be associated with a unique string token. The argument formats for all of the MediaControl calls are detailed in the source code section (see section entitled VDI Header File in Appendix).

Media Control Object Physical Structure

Each Media Control Object is a class object privately derived from an MCOPARAM (see section entitled VDI Header File in Appendix) structure. Regardless of the signal type (audio/video/image/data) represented by the Media Control Object, the MCOPARAM structure contains sub-records for all signal types. The programmer need only attend to the relevant section for the signal type for that object. There are a number of requirements as to the structure of the Media Control Objects physical structure, with regards to the specific details of its implementation.

- All member functions and member data in the Media Control Object, are protected, and can only be accessed by the PDI.
- Class PDI is a friend to all instances of class MCO.

- Class VDI cannot access any MCOs directly, except through specific members that are implemented by class PDI.
- All MCO members presented to the PDI, should be simple, device-independent operations to manipulate the settings and operations precisely outlined by the audio, video, image, and data records contained in the MCOPARAM data structure.
- Each MCO should be fully cognizant of its signal type and signal direction, and prohibit operations that are inconsistent with its fundamentally characteristic properties, i.e. cannot attach audio output to a video display.
- The handle of a new MCO must be added to the VDI tables in DEVICEPARAM when that MCO is created, and removed when deleted, such that the client always has a clear picture of available system resources.
- Events must be queued for each and every MCO operation executed.
- Regardless of the complexity of underlying system components that must be initialized, addressed, or monitored to implement Media Control Object operations, it is critical that the designer reduce the invocation of such processes to simple operations described by the Media Control Object settings.

MCO Interface to the Media Access Control Layer

Each MCO controls the device underlying the signal it represents by making requests to the Media Access Control layer components that drive them. The PDI pure virtual override DevMediaControl presents settings

to the Media Control Objects, and the Media Control Objects then go on to map the setting to a physical device control operation. FIG. 22 shows the control flow for device control operations that are presented to MCI drivers that comprise the MAC layer. This diagram has greatly simplified the pathway from the VDI to the MCI driver, eliminating most of the interactions with the Media Control Object. In short, the PDI prepares a Media Control Request Record, and presents it to the appropriate Media Control Object so that the object can fill in its fields, and present it to a corresponding MCI driver (see FIG. 22). Note that a device control operation initiated by the local station can result in the station assuming a new H.221 device mode, which is then transmitted to the remote station (if currently connected) for station synchronization, referred to as the "establishment of common modes" by Recommendation H.320. Finally, an event is added to the VCO event queue describing the new Media Control Object setting that has taken place.

STATION ATTACHMENT MECHANISM

There are a number of considerations with regards to the system components that must be created to support the "attachment" between two interconnected VMCS stations. A pathway must be established between the two stations such that they can share text string commands streams. Once this pathway is available, the two stations must come to a mutual understanding how they will interact; that is, which station is the master, and which is the slave.

Beyond the standard attachment mechanism described, a third station can control any one of a number of stations that are themselves interconnected in a conference. In this modality, a "third party"

controller, or "remote operator" can intervene in conference already in progress to assist, diagnose, or monitor one of the conferees. The details of designs that would accomplish this task are supportable by the VMCS, but beyond the scope of this disclosure. Below follows a description of the details for the VCO components used to implement the remote station attachment mechanism.

Command Stream

From the perspective of one end of the attached station pair, the command stream is bi-directional -- to and from the remote station. A Media Control Object supporting text data output to the remote station, and another Media Control Object supporting text data input from the remote station, are created to encapsulate the data pathways. Since mostly text message data will be exchanged, the pathway need only support low bandwidth (less than 16 kbit/s) on an occasional (asynchronous) basis. Data can be transported out-band on a separate channel (such as an ISDN D-Channel) or in-band, perhaps multiplexed with video data. The data transport mechanism for message data can also be accomplished through a tertiary source using an entirely separate connection from that used for primary communication. All messages to the remote station are written to the Media Control Object encapsulating the pathway to the remote station, while messages arriving from the remote station generate events as they are received by the Media Control Object encapsulating the pathway from the remote station.

Event Encoder

This component converts binary event records added to the VCO event queue to a text event message representation, and then sends it to the remote station. A definition of the binary event record is provided (see

section entitled VDI Header File in Appendix), however the exact text event message format is left to the system designer. Suffice it to say that the text message format used should be entirely universal to allow all VMCS implementation platforms to engage in "attachment". String tables in the Linguistic Controller areas of the VCO are used to convert the enumerated arguments to string tokens, whenever possible, while purely numerical arguments (such as parameters) are converted to ASCII hexadecimal strings. Each event message must minimally include the following information:

- Event identifier
- 32-Bit parameter 1
- 32-Bit parameter 2
- Source station identifier

The event encoder is usually only accessed when the VCO is attached to a remote station. While the VCO is attached, the encoder is invoked by the VCO queuing-mechanism each time an event is queued. The encoding takes place using a separate thread of execution to avoid interference with device timing.

Event Decoder

This component converts text event messages received from the remote station and converts them to binary event records that are then added to the local VCO's event queue. This process is the inverse of the encoding process, and its success depends upon the consistency of the text event message format selected. The source station identifier tells the receiving VCO where the event came from, and the string tables in the Linguistic Controller are used here to derive a numeric representation by comparison of the string token keywords to their relative position in the tables, and derive a string index when the token is identified. The event

decoding takes place using a separate thread of execution dedicated to fielding incoming command and event messages.

Command Encoder

This component converts SCI calls to a text command message representation, and then sends them to the remote station. As with the event encoder, the exact text command message format is left to the system designer, and correspondingly, it should be entirely universal for reasons said. String tables in the Linguistic Controller areas of the VCO are used to create text command messages whose format is variable depending on the SCI command encoded. The command encoding takes place using a separate thread of execution dedicated to fielding incoming command and event messages.

Command Decoder

This component redirects the text command portion of the shared messages to the local VCO terminal input port, where they are interpreted, and then used to generate calls to the local VCO's SCI, just as if they had been input from a local user at a terminal keyboard. This process is the inverse of the encoding process, and its success depends upon the consistency of the text command message format selected. The event decoding takes place using a separate thread of execution dedicated to fielding incoming command and event messages.

Determining Capabilities

Each VCO has associated with it independent parameter blocks for remote control parameters and remote monitoring parameters. There are separate parameter blocks each containing flags that indicate their current operating modes, and operating capabilities with regards

to attachment. The control and monitoring capabilities are stored as nonstandard capabilities in the local capabilities lists, and each station will know those of the other at the time of connection. Once attachment has been established, it is the station that first requests a particular control or monitoring "context" that will able to assume that role. When a role is assumed, flags in the associated parameter blocks mark the state, and prohibit any further context changes till the stations are detached, disconnected, or the change is initiated by the "controlling" or "monitoring" station.

Remote Monitoring

As a subset of a full remote control context, monitoring of a remote station requires only that the "monitor" station receives a stream of the events queued by the "monitored" VCO. Upon connection, both stations are monitoring their own locally queued events; they are operating under a local monitoring context. If one station permits (indicates it is capable of) being monitored, the other can change its monitoring context, by setting its monitor mode flags to include the remote stations events, and therefore add the events from the other station to its own event queue, in addition to continuing to monitor its own event stream concurrently. Alternately, it can monitor only those events of the other station, or monitor any one of a group of stations in a conference, as they come into focus (though a virtual attachment must be established with each individual station prior). Monitoring can occur bi-directionally at the same time; two stations may monitor each other concurrently.

Remote Control

The assertion of control by one station, over another, proceeds similarly to the establishment of a monitoring context. In doing so, one station's capabilities indicate it will assume a slave operating context, while the other will be the master. Upon connection, both stations are controlling their own systems; they are operating under a peer control context, and have no ability to control the other. The master VCO initiates the transaction to request operational control of the other, and if consent is given by the other, it assumes the role of master, the other as its slave. The slave VCO now reacts to commands sent by the master, just as if the commands had been issued by its local client application. For remote control to occur, the master must also monitor the event stream of the remote VCO. With the ability to transparently send commands to a slave station, and transparently receive events from the station in response to those operations, the VCO that assumes the master control context becomes a fully operational virtualized representation of the slave station. And client applications that run on the master VCO are (practically speaking) virtual clients of the slave VCO.

MULTIPOINT CALLS

A number of recommendations serve to define standard designs, protocols, and terms used for audiovisual connectivity sessions that involve more than just a local and remote station. ITU-T Recommendation H.231 (see ITU-I Recommendation H.231, MULTIPOINT CONTROL UNITS FOR AUDIOVISUAL SYSTEMS USING DIGITAL CHANNELS UP TO 2 Mbits/s, 1994) describes such units and their various configurations. Attendant Recommendation H.243 (see ITU-I Recommendation H.243, PROCEDURE FOR

ESTABLISHING COMMUNICATION BETWEEN THREE OR MORE AUDIOVISUAL TERMINALS USING DIGITAL CHANNELS UP TO 2 Mbits/s, 1994) describes the specific operations performed in a multipoint call environment, such as adding, dropping, and viewing specific conferees from the conference. These recommendations serve as the impetus for the logically defined multipoint control operations incorporated into the VMCS server (VCO) architecture. At time of writing, there exist fewer than a half-dozen widely available devices supporting a significant subset of the procedures outlined by the recommendations.

VCO Multipoint Control Operations

Support for multipoint operations by the VCO requires the use of a multipoint control unit (MCU). These devices have a number of NIUs, referred to as ports, that allow many audiovisual connectivity stations to attach to them concurrently -- the MCU serving primarily to bridge the signals from one station to that of many others, so as to enable a group to view an individual. A specialized algorithm determines which conferee, at any given time, is to be seen by the others. In most cases, the strength of the audio signal (voice presence) determines the individual that addresses the conference. A chairman is specified to direct the conference, and he possesses special privileges to administer its outplay; his responsibilities as chairman include the appropriate allocation of resources, the creation of conferences, and the configuration equipment as needed.

The multipoint control operations supported by the VCO MultiCall SCI Member are shown below (see FIG. 30). Calls to this MultiCall member map to the PDI DevMultiConnect member, one that provides an actual physical implementation for them. To support the

operations, this PDI member will typically be constructed to issue vendor-specific commands to a MCU resident in the station, or cabled to it with some communications link. All of the major operations needed to directly control the mechanics of a conference, as its chairman, are included. Further discussions of these operations are provided in the source code (see section entitled VDI Header File in Appendix). The CALLPARAM structure in the VCO has a number of flags and variables used to track and configure multipoint control operations. A series of flag values referred to as MULTICALLSTATES provide detailed indications as to very specific states in both the local VCO, and the conference in which it is participating.

VCO IMPLEMENTATION PROCEDURE

The procedure to implement a VCO differs depending on whether the project is to create an initial server component, or to create a new component (from an existing one) that will work on a new multimedia connectivity sub-system. While primary (initial) VCO development requires considerable effort, secondary (subsequent) VCOs are comparatively minor projects in both time and resource requirements. The development of VCO clients can proceed concurrently with, and independently of, that of the server(s); the production of VCO Clients requires markedly less design and development expertise than is required to produce the VCOs themselves.

VCO Implementation Sequence

The following procedure is applied to primary VCO development. Secondary VCO development efforts (based on the same design) reuse almost all components, without

modification, thus steps 1, 3, 4, and 5 are not necessary to create them.

1) Derive VCO Design

Prefatory to development of a VCO, a detailed design for a specific implementation must be derived from this disclosure of the VMCS Technology. In addition to describing the internal mechanism, this design provides the definitions of the external interfaces to client applications, and to other VCOs created to run on different types of host stations; these specifications should be preserved for all subsequent implementations to maintain interoperability across the network.

2) Establish Sub-system Operability

It must first be determined if the connectivity sub-system and associated devices are operational. Any test procedures and test programs must be executed on a sub-system installed and configured exactly as specified by the particular VMCS server (VCO) design discussed in step 1.

3) Develop Virtual Device Interface

Source code development for the VDI includes creating all of the classes from which it is derived. The device-independent portion of the VCO is implemented as a distinct unit that can be built without any dependencies on any device-dependent, or vendor-specific components.

4) Develop PDI Shell with Emulation Support

Source code development of the PDI proceeds initially as an attempt to create a minimal implementation that will support emulation

for its pure virtual device control overrides. Calls for device support return "Not Implemented", or if the VDI has invoked emulation mode, they return an emulated response, as if a device was physically present.

5) Debug VCO as Emulation Object
Running in emulation mode, the VCO is debugged to verify the functionality of its device-independent portions. A small sample client application must be created to invoke SCI members for testing purposes.

6) Develop PDI Physical Device Controls
Once the VCO is fully operational and debugged under emulation, physical device control is added to the PDI by providing a software linkage of the pure virtual device control overrides with the connectivity sub-system and associated devices previous emulated in software. Media Control Objects are implemented, tested, and integrated into the device control system.

7) Debug VCO as Live Device Control Component
Running in the default device control mode, the VCO is debugged to verify the functionality of all of its components and features in a "live" connectivity environment. A sample client application must used, as in step 5, to invoke SCI members for testing purposes.

DEPLOYMENT

This section describes usage of the disclosed VMCS technology in an operational configuration. Consideration is given to the underlying theories of VMCS

operating principles, including discussions relating to the sequences of operations needed to manage various multimedia connectivity tasks encountered during VCO connectivity sessions. In essence, the following section serves as an operations manual for the VMCS, focusing mostly on the services provided by the VCO.

OPERATING PRINCIPLES

CONSTRUCTION

VCO construction is a process defined by C++ as a means to create a binary software object from a class definition. With the VCO, the client initiates a construction process that is in no way different from that of any other class, and the VCOPARAM data structure is initialized, along with other initialization tasks. Upon successful construction of the Virtual Connection Object, its principle data structures and capabilities are available for perusal.

- Construction gives the client access to VCO internal data structures, use of basic device-independent services, and the results of a perfunctory examination of requisite supporting sub-components.
- By the time VCO construction has completed, its dispatcher is running, and it has uploaded all configuration parameters from backing store. At this time, the notification mechanism is fully operational, and Notifier Objects can be created for use by the client.
- Following VCO construction, the VCO terminal input and output ports are active, and can be used by the client to send text messages to output devices, or to decode command input.

- Although the VCO may have successfully constructed, no drivers have been loaded, no devices have been accessed/initialized, and there is no way for the client to know if the hardware necessary to run this VCO is actually installed.

DESTRUCTION

VCO destruction is a process defined by C++ as a means to destroy a previously constructed binary software object. With the VCO, the client invokes a destruction process that is in no way different from that of any other class. During the process of VCO destruction...

- If the VCO is connected, a disconnect is issued -- the VCO waits until the disconnect completes -- and all associated VCO components and drivers are unloaded.
- The VCO dispatcher is halted, and any Notifier Objects are deleted, thus Notification Receiver Objects in the client will no longer receive information.
- The terminal, and all other VCO services, are no longer available for client use, since the VCO does not exist following destruction.

NOTIFICATION

Once a client constructs a VCO, it can create a number of Notification Objects, the maximum of which is determined by the system designer. Notification indications are sent to the client immediately following construction, should any triggering events should occur.

- The NewNotifier command enables the creation of a Notifier Object, returning the handle to one just created as specified. When the Notifier Object is created, it is intimately

associated with one particular Notification Receiver Member contained within one particular Notifier Receiver Object, neither of which can be changed; a new Notifier Object must be created to direct notification to a new object-member combination.

- The DeleteNotifier command can be used to delete Notifier Objects, the handle of the Notifier Object being used to identify the instance to delete.
- The EnableNotifier command can be used to enable or disable the specified Notifier Object. Each Notifier Object can be disabled to stop the VCO notification process to that particular Notifier. When enabled and actively receiving notifications, a call to the client's Notification Receiver Object (by the Notifier Object) occurs to prosecute indication of the occurrence of an event, during which time, it cannot be reentered by another such call until it returns from processing that event currently in play.
- The SetNotifierTriggers command allows the client to change the set of events that cause the Notifier Object to trigger; that is, invoke the Notifier Object to deliver information to a specific member function of a specific class object, indicating that a particular VCO event has taken place.
- The TriggerNotifiers command can be used to trigger one particular Notifier Object, unconditionally, or to present an event to the triggering mechanism, allowing it to trigger all Notifier Objects for the VCO that

contain that specific event in their triggering profile.

- Notifier Objects cannot be created or deleted while processing the notification of an event, because the internal list of Notifier Objects is locked. However, the triggering profile for the Notifier Object can always be modified.

CONFIGURATION/SYSTEM SETUP

There are two distinct services available to VMCS system users for configuration/setup. The first is the ability to maintain a VCO initialization file that stores a text record describing all of the startup defaults for major categories of VCO settings, including a description of its network service, standard terminal output device, local station identity, dispatcher rate, device and connection time-outs, conference profile, and other default settings. The second is the ability to invoke dialog boxes containing detailed configuration/system setup utilities that are provided for each of the four possible media types (audio/video/image/data) that are potentially handled by the VCO.

- The initialization file is read at VCO construction time, and its user-readable text arguments are converted to an internal binary format stored in the VCO, accessible as a data structure to VCO clients.
- The SetConfig command allows clients to write a new configuration structure to the internal VCO configuration structure, and at the same time activate the new settings.
- The RefreshConfig command allows clients to upload the VCO's internal configuration from its default initialization file, and at the

same time activate the new settings. Alternately, the command can be used to upload the internal configuration stored in the initialization file into a client configuration record, leaving that of the VCO configuration record unmolested.

- The StoreConfig command enables clients to store a configuration record directly in the default VCO initialization file, overwriting the existing configuration, or alternately, it enables clients to similarly store a configuration record held privately by the client. In either case, the data elements in the configuration record are converted to user-readable text arguments prior to being stored in the initialization file.
- Integral configuration utility screens enable end users to adjust relatively minor vendor-specific device driver and operating specific system parameters that do not map well to the generalized, device-independent controls offered by the VDI and associated media control device control settings.
- The VMCS concept intends these adjustments to be limited to those settings that are usually set once during initial system installation, and subsequently left mostly alone; they are settings that tune and enhance the operation of the standard VCO device control operations, and are not intended to duplicate or replace them.
- The strategy behind these utility screens is identical to that used by advanced microcomputer operating systems, employing graphical user interfaces, whose printer

device driver designs require an integral "setup dialog" to enable the user to configure the specialized hardware features of a target printer device, prior to sending the job to the print queue.

- As specified by the system design, configuration and setup parameters adjusted in these utility screens are used to set the device default settings that are later manipulated through standard SCI calls to the device control sub-system. Moreover, vendor-specific configuration files are modified here via links to their particular private software component configuration scheme.
- The SetupAudioDevices command invokes the audio setup utility, which provides adjustments for microphone input sensitivity, frequency equalization, output gain, specialized physical input/output port selection, noise filtering, and recording options.
- The SetupVideoDevices command invokes the video setup utility, which provides camera adjustments such as white balance, access to test modes, NTSC/PAL mode selection, and focusing mode selection. Additional adjustments for video display include those that affect color, tint, hue, brightness, horizontal alignment and vertical alignment.
- The SetupImageDevices command invokes the image setup utility, which includes output settings for any hardcopy display/printer device, or video display adjustments for factors similar to those in the video setup.

Input settings for imaging devices typically relate to form size and ambient lighting.

- The SetupDataDevices command invokes the data setup utility, which is more nebulous in its specific format, and whose settings may range from communications port settings to disk drive specifications for backing store. The VMCS make no presumptions as to the ultimate use of data streams, thus the derived VMCS design must specify the data setup utility. Typically, a VCO that directs a data stream to/from a communications port will maintain settings for baud rate, parity, stop bits, and other asynchronous data transmission settings.

TERMINAL SERVICE

The VCO terminal service provides an input and output port. The terminal output port functions as a standard output device that displays character stream data written to it, while the terminal input port accepts character stream data written to it, and interprets it as VCO text commands. Character stream data takes the form of null-terminated ASCII strings, referred to as text messages. The null character is used to denote the end of the message. Format dictates VCO text command strings terminate with a carriage return, and intervening nulls that terminate command (sub-strings) are ignored by the decoder.

- Text messages sent to the terminal output port may be written, concurrently by the VCO, to more than one physical output device, following each client text output operation. Physical output devices configured to be written when text messages are sent to the

VCO terminal output port, are referred to as attached (to the output port). Resultantly, clients sending text messages to the default VCO terminal output port will find that the same text message has been duly written to all output devices attached to the terminal output port.

- Text messages sent to the terminal input port are parsed into string command and argument tokens, and interpreted by the VCO as representations of SCI commands. Once a text command has been fully decoded, it is used to affect SCI member functions, thereby providing a scripted control mechanism to any VCO client; scripts can be generated by the local client, read from script file, transmitted from a remote client across a connection, or entered directly from a terminal.
- The terminal service is available immediately following construction. A default output device is identified in the configuration stored in the VCO initialization file, and attached to the terminal output port.
- A range of standard output device types can be added or removed from the list of output devices attached to the output port. All output devices are written with every text message that is sent to the terminal output port.
- The ToTerminal command is used to send an optionally formatted text message to the VCO terminal output port. The command functions similarly to "print" statements used by

various programming languages that send text to a standard output device.

- The FromTerminal command is used to send an optionally formatted text command (VCO script command) to the VCO terminal input port. NOTE THAT the terminal input port cannot be read for input by the client, as the term input refers to the provision of input data to the VCO from the client. The client is the source of the character stream. Since the VCO has no reason to request commands from the client (only the client can initiate the issuance of commands) the onus is on the client to send those commands to a mutually agreed-upon place where the VCO can receive them, and in so doing, invoke the VCO to decode them. Reiterated more plainly, the client "stuffs" script commands in a buffer, and calls the VCO to interpret them.
- A Notifier Object may be created and utilized as a terminal output device. When the Notifier Object is attached to the terminal output port, it may be triggered by any VCO (or client) text output sent to the terminal output port, thereupon the Notifier Object is explicitly triggered so as to make the client's Notification Receiver Object the recipient of every text message sent to the terminal. This mechanism allows any client to direct all text message output to a client's text processing routine.

NETWORK SESSION

Establishing a network session is accomplished by invoking a sequence of SCI members. Construction and

destruction of the VCO frame the connectivity session in that a VCO is usually selected and constructed just prior to connecting to a remote station, and is subsequently destructed immediately following the termination of the connection to it, therein freeing all system resources erstwhile allocated to the maintenance of that association. The process of associating two stations across the network, for the purpose of exchanging audio, video, image, and binary information between them, is advanced by a sequence of VCO operations next described:

- The Open command initializes the encapsulated multimedia connectivity sub-system, loading and starting all supporting vendor-specific software components. Until the "open" is performed, only non-device related VCO services are active. All devices are started and tested to determine that they are fully operational. All available signals are represented in newly created Media Control Objects, and then are opened for use. The entire sub-system, including all hardware and software components, is set to default startup settings, and the network connection is verified. If the open is successful, a connection can be established at any time, and all local devices are accessible to the client, to be controlled by the user. Incoming calls from a remote station may be handled following a successful open.
- The Call command initiates a call to a remote station. In the preferred VMCS embodiment, the network is the ISDN (telephone system) and the "call" operation results in a direct dialing of the number of the remote stations line(s). Just as with a standard telephone,

the visual telephone service provided by the VMCS requires no further action by the client, and a simple result is returned. A successful connection process results in the execution of an internal VCO process to establish a series of baseline operating modalities for the type of session established. For the visual telephone, a video window is displayed showing the far end, remote station audio is audible, and both local video and local audio are sent to the remote station. Image and binary data facilities are initialized as idle, and pathways await client operations to exchange information. In short, the "call" operation of the VCO's visual telephone system works in an identically analogous fashion to making a "call" with a standard analog telephone: dial, connect, then concurrently exchange information bi-directionally, without delay.

- The MultiCall command enables the initiation of a number of complex multipoint control operations (see FIG. 30). If the local station is the conference chairman, the VCO client can add and remove conferees from the conference, among other administrative functions, all of which require the ability to control a multipoint control unit (in some direct or indirect fashion according to the actual physical implementation of the VCO service). Other multipoint operations include various query and broadcast operations that may or may not require an advanced level of MCU control. The client can query any of these operations to determine if they are

currently supported by the session in progress. The client can determine if the VCO implementation supports the operations at all by examining the VCO capabilities list, which includes multipoint control capabilities that are proprietary to VMCS technology.

- The Hangup command enables the client to drop one, or more (all) lines used for the current call. Similar to the "call" operation, the "Hangup" operation of the VCO's visual telephone system works (by default) in an identically analogous fashion to the "Hang-up" of a standard analog telephone: end the call without delay.
- The Close command shuts down all devices in the multimedia connectivity sub-system, and unloads all vendor-specific software components. All client access to device control services is no longer available, and Media Control Objects are all destructed. Neither incoming, nor outgoing calls may be handled, and only the non-device related services remain active.

MEDIA DEVICE CONTROL

Device control is available to the client by the manipulation of Media Control Objects via calls to the MediaControl SCI member. The VCOPARAM structure contains a list of the names of all available Media Control Objects active in the system. This list will be empty if the VCO has not been opened first, as the list of Media Control Objects reflects the signals available for manipulation by the client. A list of the handles to these Media Control Objects is also available, and information about them may be obtained using the

appropriate SCI member function. Principles governing the control of the devices in the encapsulated sub-system, and their associated operations, are shown below:

- There are four signal types (audio, video, image, data) and four signal directions (to remote, from remote, to device, from device) from which sixteen Media Control Object type permutations are derived. These are the sixteen default Media Control Object types that may be addressed by type in operations (see FIG. 7A).
- Following successful completion of a VCO open command, any available signals in the VCO are represented as Media Control Objects, automatically opened for use, and enabled. They are not switched on initially, but must explicitly be turned on by a client command or by connection to a remote station.
- The MediaControl command enables clients to change the settings for any active (existing and enabled) Media Control Objects assigned to be one of the sixteen default types. Additionally, switching of signals (in the form of plugging together source and destination Media Control Objects), creating composite signals from multiple input signals, and a host of related functions can also be accomplished with this command (see section entitled VDI Header File in Appendix).
- The SetDefaultMco command can be used to assign a non-default Media Control Object as a default Media Control Object, if it is the same type as the one it is replacing.

- The GetMcoHandle command can be used to retrieve the handle to an Media Control Object from its name label or object type. GetMcoLabel is a command that allows the name label for the Media Control Object to be determined from its handle.
- The GetMcoParam command can be used to retrieve the internal parameters and settings for a specified Media Control Object, and thus it can be used to determine the operational states and settings of the signal represented by the Media Control Object, as well as the device (if any) that is the source or destination of that signal.

BINARY DATA OBJECT EXCHANGE

Data objects may be exchanged with single operations, between stations that are both running a VMCS; each of which fully understands the data formats and "transport layer" protocols of the other end. For now, such issues are left to resolution by the system developer who must determine the exact protocol used to transfer the VCO's data objects as described herein. Though lacking in international standardization, manipulation of binary data objects is operationally well defined and described to client applications by the VMCS (The Software Control Interface and the logical manipulation of which is clearly implemented in the "session layer" residing in the VDI). Fortuitously, the ITU is currently working on Recommendation T.120 (Data Conferencing) to enable standards-based exchange of binary data objects. A case could be made for the utilization of this VMCS model for all connectivity software systems on the sole basis of its ability to insulate applications from the ongoing T.120

specification, and the complexity of its implementations. As of this writing, T.120 remains incompletely specified, only partially implemented by any real products, and even less well understood by system developers. It is expected that T.120 will eventually provide the "language" necessary for the VCO to conduct its "binary data object exchanges" below the "session layer", in an entirely standards-based fashion.

If the remote station is not running a VMCS, simple data buffers and cursor positions may be sent according to existing procedures for information sharing in H.320, but support to transfer entire binary and/or text files may not be available. If it is determined that the remote station connectivity sub-system is a compatible VCO (using the IsVCO command), then any data object can be transferred. Otherwise, if the data object to be transferred is a file, the remote station will be unable to respond to the VMCS proprietary file transfer protocol used on the data channel. Support for the exchange of cursor positions, facsimiles, still pictures (images) and raw data buffers is supported by most H.320 compliant stations, and thus possible between a VMCS and any remote station to which it may be connected. The mechanics of exchanging data objects between stations are discussed below:

- The TransferBuffer command enables a client to send a buffer of binary data through the data channel (or multiplexed data channel using an allocated portion of its total bandwidth), to the remote host. This command can also be used to determine if the data channel to the remote station is available.
- The TransferObject command enables a client to send or retrieve a specified data object

through the data channel, or multiplexed data channel.

- The transfer operations specify a Media Control Object whose data signal is used to transport the data. If the remote station is running a VMCS, the direction of Media Control Objects signal determines whether the transfer operation sends local data to remote station (data to remote) or is a request to retrieve data from the remote station (data from remote).
- The Media Control Object used for the transfer contains data structures and variables that describe the actual status of the transfer.

SYSTEM INFORMATION

Access to system information is highly restricted, from the view of the client, and is only available through SCI members. These members handle a wide variety of VCO states and parameters, and provide that information to the client in divers formats:

- A number of VCO states and conditions are reported by SCI members returning boolean results. These boolean test members allow clients to determine if the VCO is ready to make a call, if a call is currently being setup, if a call is connected, if a multipoint call is in progress, if the remote station is a VCO (running a VMCS), or if the current VCO exists in more than one instance.
- References to copies of data structures, stored internally in the VCO, are returned for those specific categories of information relating to devices, configuration, call,

protocol, control, and monitoring parameters. One member returns a reference to a copy of the entire VCO system information data structure.

- The internal data structure of a Media Control Object can be accessed in a way similar to other data structure, with the client specifying the default Media Control Object type, or a handle to one. Also, the handle for a Media Control Object can be obtained from a Media Control Object label or the Media Control Object's type.
- Text names for most enumerations, constants, and system objects can be retrieved from the VCO using any one of a number of members designed to return labels to them.

PROTOCOL MANAGEMENT

The H.320 protocol defines the basic operational structure of the VCO's multimedia connectivity services, and from the standpoint of the client, is mostly transparent to its functionality. An exception lies in the VCO support for the manipulation of device modes and capabilities; it is useful for the client to affect the system's capability list, as well as the set device modes directly. Moreover, such access allows more knowledgeable clients to perform advanced, or less well supported operations at a low-level.

- A data structure in the VCO, referred to as PROTOCOLPARAM, provides the client with information about the H.221 multimedia connectivity protocol. A full accounting as to the progressions of which, is provisioned by this useful data structure, specifically with regards to current and pending device

modes for audio, video, data, and miscellaneous operations.

- The SetConfProfile command enables the client to specify a conference profile that describes a preferred set of audio, video, and data modalities (relative to the available bandwidth of the connection) that define the overall quality of the connectivity session.
- The SetModes command enables the client to specify one, or more, H.221 device modes by presenting them to the connectivity sub-system. This command is used in conjunction with the VerifyBandwidth command to determine if there is sufficient bandwidth available in the connection to support a specified set of audio and data modes, while retaining the current video mode.
- The SendCaps command enables the client to transmit its entire H.221 capability list to the remote station.
- The SetDeviceTimeout enables the client to specify the number of milliseconds the Network Interface Unit should wait for a response from a network request before timing out, whereas the SetConnectionTimeout enables the client to specify the number of milliseconds the system should wait for a connection to a remote station to complete prior to timing out.

VCO CONTROL OPERATIONS

A large group of operations enables the client application to adjust, control, and invoke special features of the VCO. Some of these operations enable the

manipulation of internal VCO settings that are typically left to their default settings for most sessions. A number of commands are used by a client to attach to a remote VCO for the purposes of remote control and/or remote monitoring of it.

- The EnableVco command is used by the client to alter the state of the VCO's "enable" flag, a task usually reserved for recovery from an exception that previously disabled it. The SetVcoExceptMode is used to set the exact modality used by the VCO to handle exceptions.
- The SetVcoTraceMode command is used to instruct the VCO as to exactly which operations and components should be configured to direct trace information to the VCO terminal output port.
- The EnableMultiCallOps command is a simple switch that is used to select the client's accessibility to the multipoint control operations. To disable these operations causes them to return "disabled" to the caller.
- The EnableDispatcher command is used by clients to pause the dispatching of events from the VCO event queue. This operation is used when the client wishes to "idle" the VCO, while allowing underlying devices to function as best as they may. The related command SetDispatcherRate enables the client to change that rate at which events are dispatched from the VCO event queue, a task usually performed when a faster or slower event stream is required by the client application; faster dispatching rates allow

the client to operate at speeds closer to those of the encapsulated multimedia connectivity sub-system.

- The UpdateCapsList command is used by the client to add or remove a device capability to the VCO device capability list, a version of which is transmitted to the remote station during the connection process. A related command, UpdateModeCapsXRef, allows the client to add or remove a mode-capability cross-reference record that is used when the VCO attempts to establish common operating modes with its remote station peer.
- The EmuControl command enables the client to access an internal VCO emulation facility. Features include enabling/disabling the VCO device emulation mode, and invoking predefined emulation sequences.
- The AttachToRemote command is used by the client to provoke the VCO to attach to its remote station peer, if that peer is a VCO (running a VMCS). The DetachFromRemote command eliminates any attachment between interconnected VCO peer stations. When the VCO is attached, the SetVcoControlMode command is used to select the master, slave, or peer modality of operation for the local station, with respect to the remote station.
- The SetVCOMonitorMode command enables the client to select the event stream for the VCO to process. Events from the local station, an attached station, a group of stations, or some combination of station are directed into its event queue for subsequent dispatching.

- The SetStationLabel command is used to assign a text label to the local station so that it may be referenced (locally or by a remote station) by that name.

SAMPLE CLIENT APPLICATION

Source code in this disclosure (see section entitled Sample VCO Client Application in Appendix) illustrates the use of VCO services by a multithreaded, event-driven VCO Client application. This simple program does not utilize a graphical user interface, but directs its output to the standard output console. The program examines a VCO's capabilities to determine if it supports required audio, video, and data modes, and opens a connectivity session if it does. Source code is also provided for the many header files used by both the VCO Clients and the VCO itself.

The invention is meant to cover all of the above-mentioned alternative approaches as well as others not specifically mentioned. The above-mentioned embodiments and others are within the following claims.

SOURCE CODE

VDI HEADER FILE

```
/*-----------------------------------------------------------------------
                    VIRTUAL DEVICE INTERFACE HEADER FILE
                                  for
                VIRTUALIZED MULTIMEDIA CONNECTION SYSTEMS

                                ABSTRACT

This source module contains definitions for the principle software enumerations, constants, data structures, and member functions
that comprise the Virtual Device Interface (VDI) software component of a Virtual Connection Object (VCO). These items must be
incorporated into both the client and server entities of any VMCS implementation, in some form of computer language
representation. The device interface components are internal (non-public) to the VCO, and are of the pure virtual type. All other
member functions, structures, and constants shown below are used by every VCO to enable structured access to their encapsulated
multimedia connectivity sub-system, and by VCO Clients desirous of structured access to a device-independent representation of the
same. These member functions and member data objects are collectively referred to as the Software Control Interface; they are the
same for every VCO implementation, thus enabling creation of device-independent connectivity applications that exploit their
services.

:SOURCE FILE.      VDI.H)

                            PROGRAMMING NOTES

1. This module contains only C++ source code and structured comments using the " // " notation to denote comments (in addition
to the standard C comment notation using " /* */ ").

2. The term unknown refers to a value, condition, or requested operation that can not be identified; that is the usage of this word
connotes a patently errant condition.

3. The term unexpected refers to a value, condition, or requested operation that is identifiable, but is inappropriate given the current
set of preconditions; that is the usage of this word connotes inappropriateness of context.

4. The term exception refers to an occurrence of a severity that warrants abandonment of the connectivity sub-system (a fatal error);
such an occurrence connotes significant destabilization of the VCO has occurred and further usage risks a system crash.

5. The term blocking describes calls that wait for the requested operation to fully complete, the return value of which indicates its
results. This modality of operation is the default for all calls. If there is a "_IsBlocking" argument to the call, and it is set to 0
(false), then the call returns immediately without waiting for the operation to complete, typically returning "pending" if the request
is valid. Indication as to the result of this operation comes from the insertion of a descriptive event into the VCO event queue upon
its completion.

6. All character pointers (char*) point to null-terminated ASCII strings of a length less than 256 characters, including the null.

7. The term label refers to a string as defined in (6.) above, except that it may not contain spaces and its length is less than 32
characters, including the null.

                        ARGUMENT SYNTAX
                                for
                    VIRTUAL CONNECTION OBJECT EVENTS

Notification of the occurrence of a standard Virtual Connection Object event is initiated when a notification object in the host VCO
"triggers", and subsequently calls a specific event handling function residing in a designated Notification Receiver Object (NRO);
that is, a software object that contains member functions implemented specifically to respond appropriately to that type of system
event.

EVENT IDENTIFIER    PARAMETER 1                  PARAMETER 2
---------------     ---------------              ---------------
NullEvent           Don't care                   Don't care
NewEmuState         True if emulation enabled    Don't care
NewEmuOp            < EMULATIONOP >              Don't care
NewRefCount         Previous reference count     New reference count
NewDeviceState      < DEVICESTATE >              Device index
NewMcoFocus         < MCO_TYPE >                 Ptr to media ctrl obj label
NewLocalCaps        Previous number capabilities New number capabilities
```

```
                NewRemoteCaps       Previous number capabilities    New number capabilities
3550            NewRcvMode          < BASCODE >                     Don't care
                NewXmtMode          < BASCODE >                     Don't care
                NewRejMode          < BASCODE >                     Don't care
                NewAudioSetting     < MCO_SETTING >                 Parameter for setting
                NewVideoSetting     < MCO_SETTING >                 Parameter for setting
3555            NewImageSetting     < MCO_SETTING >                 Parameter for setting
                NewDataSetting      < MCO_SETTING >                 Parameter for setting
                NewCallState        < CALLSTATE >                   Don't care
                NewLine1State       < LINESTATE >                   Don't care
                NewLine2State       < LINESTATE >                   Don't care
3560            NewConfProfile      < CONFPROFILE >                 Don't care
                NewDiscStatus       < RESULTCODE >                  Don't care
                NewMultiCallState   < MULTICALLSTATE >              Don't care
                NewMultiCallOp      < MULTICALLOP >                 Don't care
                NewDataXferState    < XFERSTATE >                   Don't care
3565            NewRcvBuffer        Number of bytes received        Ptr to media ctrl obj label
                NewXmtBuffer        Number of bytes transmitted     Ptr to media ctrl obj label
                NewRcvObject        < MCO_XFEROBJ >                 Ptr to media ctrl obj label
                NewXmtObject        < MCO_XFEROBJ >                 Ptr to XferObject
                NewVcoState         < VCOSTATE >                    Ptr to XferObject
3570            NewCursorPos        High-word is X, low-word is Y   Don't care
                NewTermInput        Number of bytes transmitted     True if relative to previous
                NewTermOutput       Number of bytes transmitted     Ptr to message string
                NewResultCode       < RESULTCODE >                  Ptr to message string
                ____________        ____________                    Ptr to invoking cmd string
3575            ----------------------------------------------------------------------------------------------------*/

                // BASE DATA TYPES USED IN ALL VCO SOURCE MODULES
                typedef   unsigned  char    BYTE;       // 8-bit unsigned value (standard octet)
                typedef   unsigned  int     WORD;       // 16-bit unsigned value
3580            typedef   unsigned  long    DWORD;      // 32-bit unsigned value
                typedef   const     DWORD   BASCODE;    // 32-bit unsigned H.221 bit-rate allocation signal constant type
                typedef   const     DWORD   EVENT;      // 32-bit standard VCO event identifier type
                typedef             DWORD   HNOTIFIER;  // 32-bit handle used to reference signal objects
                typedef             DWORD   HMCO;       // 32-bit handle used to reference media ctrl objects
3585
                // IMPLEMENTATION-DEPENDENT CLASSES DEFINED ELSEWHERE
                class XferObject;                       // Base class for all transfer object descriptors

                /*----------------------------------------------------------------------------
3590            32-BIT STANDARD VIRTUAL CONNECTION OBJECT EVENT IDENTIFIERS
                ----------------------------------------------------------------------------*/

                EVENT NullEvent         = 0x00000000; // NO-OP to event processor
3595            EVENT NewEmuState       = 0x00000001; // New VCO emulation state
                EVENT NewEmuOp          = 0x00000002; // New emulation operation
                EVENT NewRefCount       = 0x00000004; // New VCO reference count
                EVENT NewDeviceState    = 0x00000008; // New media control device state
                EVENT NewMcoFocus       = 0x00000010; // New "current" media ctrl Object has been set
3600            EVENT NewLocalCaps      = 0x00000020; // New local capability list available
                EVENT NewRemoteCaps     = 0x00000040; // New remote capability list available
                EVENT NewRcvMode        = 0x00000080; // New device mode set by remote station
                EVENT NewXmtMode        = 0x00000100; // New device mode set by local station
                EVENT NewRejMode        = 0x00000200; // Attempt to set device mode rejected
3605            EVENT NewAudioSetting   = 0x00000400; // New setting for audio object
                EVENT NewVideoSetting   = 0x00000800; // New setting for motion-video object
                EVENT NewImageSetting   = 0x00001000; // New setting for imaging object
                EVENT NewDataSetting    = 0x00002000; // New setting for bitstream object
                EVENT NewCallState      = 0x00004000; // New call state
3610            EVENT NewLine1State     = 0x00008000; // New line 1 state
                EVENT NewLine2State     = 0x00010000; // New line 2 state
                EVENT NewConfProfile    = 0x00020000; // New conference profile for call
                EVENT NewDiscStatus     = 0x00040000; // New disconnect status from network
                EVENT NewMultiCallState = 0x00080000; // New multipoint call state
3615            EVENT NewMultiCallOp    = 0x00100000; // New multipoint call operation complete
```

```
        EVENT NewDataXferState    = 0x00200000; // New data transfer state
        EVENT NewRcvBuffer        = 0x00400000; // New data buffer receive complete
        EVENT NewXmtBuffer        = 0x00800000; // New data buffer transmission complete
        EVENT NewRcvObject        = 0x01000000; // New data object receive complete
3620    EVENT NewXmtObject        = 0x02000000; // New data object transmission complete
        EVENT NewVcoState         = 0x04000000; // New global VCO state
        EVENT NewCursorPos        = 0x08000000; // New cursor position from remote station
        EVENT NewTerminput        = 0x10000000; // New text message to VCO (to VCO terminal input port)
        EVENT NewTermOutput       = 0x20000000; // New text message from VCO (to VCO terminal output port)
3625    EVENT NewResultCode       = 0x40000000; // New result code from interpreted VCO command
        EVENT Reserved            = 0x80000000; // Reserved implementation-dependent event

        /*---------------
         NUMERICAL CONSTANTS
3630     ---------------*/

        const int  MaxDevices     = 2;          // Max number encapsulated devices
        const int  MaxObjForDev   = 3;          // Max number media ctrl objects per device
        const int  MaxMcoType     = 16;         // Max number of media ctrl object types
3635    const int  MaxXRef        = 3;          // Max number mode-cap refs per record
        const int  MaxModes       = 100;        // Max number total H.221 device modes
        const int  MaxCaps        = 100;        // Max number total H.221 device capabilities
        const int  MaxLines       = 2;          // Max lines manageable by call controller

3640    /*---------------
         ENUMERATED CONSTANTS
         ---------------*/

        // VIRTUAL CONNECT OBJECT GLOBAL OPERATIONAL STATES
3645    typedef enum {
          VcoOpen,                              // VCO is initialized and operational for calling
          VcoClosed,                            // VCO is not operational; no calls possible
          VcoFailed,                            // VCO experienced failure, but is stmnl operational
          VcoDisabled,                          // VCO has been disabled and is no longer operational
3650      VcoStateEnd
        } VCOSTATE;

        // EXCEPTION HANDLING MODALITY FLAGS
        typedef enum {
3655      ExceptModeDebug     = 0x01,           // True enables output debug info in msg box for exception
          ExceptModeUser      = 0x02,           // True enables output "user" info in msg box for exception
          ExceptModeTerm      = 0x04,           // True enables sending exception info to terminal devices
          ExceptModeNotifier  = 0x08,           // True enables reporting of exception by triggering notifier
          ExceptModeAbort     = 0x10            // True enables abort of ops. and disables VCO on exception
3660    } EXCEPTMODE;

        // TRACE OUTPUT MODALITY FLAGS
        typedef enum {
          TraceModeDevice     = 0x01,           // True enables all low-level device trace output
3665      TraceModeNotifier   = 0x02,           // True enables notification event trace output
          TraceModeMCO        = 0x04,           // True enables media ctrl object trace output
          TraceModeCall       = 0x08,           // True enables high-level call control trace output
          TraceModeLine       = 0x10,           // True enables low-level call and line state trace output
          TraceModeProto      = 0x20            // True enable all protocol trace output
3670    } TRACEMODE;

        // VCO CONTROL MODALITY FLAGS
        typedef enum {
          CtrlModePeer        = 0x01,           // True sets local direct local access possible
3675      CtrlModeMaster      = 0x02,           // True sets local as master to control remote VCO
          CtrlModeSlave       = 0x04,           // True sets local as slave to remote VCO
        } CONTROLMODE;
```

```
// VCO MONITOR MODALITY FLAGS
typedef enum {
  MonModeLocal      = 0x01,    // True sets monitoring to include local station
  MonModeRemote     = 0x02,    // True sets monitoring to include remote station
  MonModeArray      = 0x04     // True sets monitoring array of stations in conference
} MONITORMODE;

// TERMINAL OUTPUT DEVICES FOR ATTACHMENT TO VCO TERMINAL OUTPUT PORT
typedef enum {
  TermODevNotifier  = 0x01,    // Notifier as terminal output device
  TermODevFile      = 0x02,    // File, or file system std device, as terminal output device
  TermODevStream    = 0x04,    // System data stream as terminal output device
  TermODevMCO       = 0x08     // media ctrl Object as terminal output device
} TERMODEV;

// VCO EMULATION OPERATIONS
typedef enum {
  DisableCallEmuMode,          // Disable VCO call emulation mode
  EnableCallEmuMode,           // Enable VCO call emulation mode
  SetCallDstStation,           // Set remote host as a user-station
  SetCallDstMcu,               // Set remote host as an MCU
  Exception,                   // Emulate fatal VCO exception (recoverable in this case)
  Line1Disc,                   // Emulate disc on line 1
  Line2Disc,                   // Emulate disc on line 2
  RandomLine1Disc,             // Emulate disc on line 1 at random time (w/in 1 min)
  RandomLine2Disc,             // Emulate disc on line 2 at random time (w/in 1 min)
  Line1Ring,                   // Emulate ringing on line 1
  Line2Ring,                   // Emulate ringing on line 2
  Line1Ringback,               // Emulate ringback on line 1
  Line2Ringback,               // Emulate ringback on line 2
  Line1Connect,                // Emulate connect on line 1
  Line2Connect,                // Emulate connect on line 2
  OneLineIncoming,             // Emulate 1 line incoming call
  TwoLineIncoming,             // Emulate 2 line incoming call
  OneLineOutgoing,             // Emulate 1 line outgoing call
  TwoLineOutgoing,             // Emulate 2 line outgoing call
  OneLineOutgoingBusy,         // Emulate 1 line outgoing call to busy remote
  TwoLineOutgoingBusy,         // Emulate 2 line outgoing call to busy remote
  OneLineOutgoingRej,          // Emulate 1 line outgoing call that is rejected by remote
  TwoLineOutgoingRej,          // Emulate 2 line outgoing call that is rejected by remote
  TwoLineFullCallThenDiscRqst, // Emulate 2 line call to connect, the disc rqst by remote
  OneLineAudioOnly,            // Emulate 1 line audio-only call
  OneLineAudioVideo,           // Emulate 1 line audio-video call
  TwoLineAudioVideo,           // Emulate 2 line audio-video call
  TwoLineAudioVideoData,       // Emulate 2 line media ctrl call
  CallEmulationOpEnd
} EMULATIONOP;
```

```
        // MULTIPOINT CONTROL OPERATIONS (ITU-T H.231, ITU-T H.243)
        typedef enum {
          SetConfFocus,                 // Set conference focus to specified station
3730      QueryConfFocus,               // Determine station currently in conference focus
          SetConfChair,                 // Set conference chairman
          QueryConfChair,               // Determine current conference chairmen
          AddStation,                   // Add station to conference
          RemoveStation,                // Remove station from conference
3735      BroadcastAudio,               // Enable/Disable broadcast of local audio to conferees
          BroadcastVideo,               // Enable/Disable broadcast of local video to conferees
          BroadcastData,                // Enable/Disable broadcast of local data to conferees
          GetNumStations,               // Get number of conferees
          GetStationList,               // Get list of conferees
3740      GetStationCaps,               // Get list of conferee capabilities
          GetStationAudio,              // Get audio from particular conferee
          GetStationVideo,              // Get video from particular conferee
          GetStationData,               // Get data from particular conferee
          GetStationIdentity,           // Get numbers and (if possible) label for remote station
3745      MultiCallOpEnd
        } MULTICALLOP;

        // VCO UNIVERSAL RESULT CODES
        typedef enum {
3750      Failure,                      // Operation failed for some unspecified reason
          Success,                      // Operation completed successfully
          Pending,                      // Operation is pending; standby for completion
          TimedOut,                     // Operation timed out
          Redundant,                    // Operation sets mode or value that is already in force
3755      RequestDenied,                // Operation possible, but denied for some reason
          NotImplemented,               // Operation is not yet implemented, but is forthcoming
          NotSupported,                 // Operation is not supported by this implementation
          ProcessTerminated,            // Operation depends on process that has been terminated
          Capable,                      // System capable of requested operation/configuration
3760      Incapable,                    // System not capable of requested operation/configuration
          MustBeOpened,                 // Specified object must be opened prior to operation
          MustBeClosed,                 // Specified object must be closed prior to operation
          Disabled,                     // Specified function disabled to prevent further system corruption
          InUse,                        // Specified object resource is in use by another process
3765      QueueEmpty,                   // Queue is empty (no removable objects available)
          QueueFull,                    // Queue is full (no more objects can be inserted)
          MemoryAllocError,             // Memory could not be allocated to support operation
          ResourceAllocError,           // Dependent resource could not be allocated due to error
          InternalError,                // Some unexpected serious internal error was detected
3770      TimerFailure,                 // Could not configure timer to modulate processing
          UndefinedResult,              // Operation result indeterminate; don't know what happened
          InvalidStation,               // Specified station has invalid spec, or is for some reason unknown
          InvalidDataType,              // Data specified for arg is of wrong type: such as a null ptr
          InvalidDeviceReturn,          // Return code from device driver is unexpected or unknown
3775      InvalidOperation,             // Enumerated operation/event is unknown
          InvalidOperationNow,          // Enumerated operation/event is known, but unexpected at this time
          InvalidCapability,            // Specified capability is unexpected or unknown
          InvalidMode,                  // Specified mode is unexpected or unknown
          InvalidLine,                  // Specified line is unexpected or unknown
3780      InvalidNotifier,              // Specified notifier is unknown
          InvalidObject,                // Specified object is unknown
          InvalidSetting,               // Specified setting is unknown for this object
          InvalidParam,                 // Specified parameter is unknown for this setting
          CmdSyntaxError,               // Syntax error in "command" portion of message
3785      ArgSyntaxError,               // Syntax error in "arg" portion of message
          NotEnoughBandwidth,           // Not enough bandwidth for requested operation
          CallMustBeConnected,          // Operation only possible while connected to remote station
          NoCallForLineAdd,             // Attempt to add unknown conferee
          LineIsDown,                   // Line has disconnected
3790      LineConnectFailed,            // Line connection failed
          LineNotConnected,             // Line has not yet fully connected
          LineIsBusy,                   // Line is busy
          DisconnectRequest,            // Line disconnect is requested
```

```
// DISCONNECTION RESULT CODES FROM NETWORK LAYER
  DiscStatusUndefined,                  // Disc status indicates undefined condition
  DiscStatusNormal,                     // Disc status indicates normal
  DiscStatusProtocolError,              // Disc status indicates protocol error
  DiscStatus_0_PrefixNotAllowed,        // Disc status indicates zero prefix is not allowed
  DiscStatus_1_PrefixNotAllowed,        // Disc status indicates one prefix is not allowed
  DiscStatus_1_PrefixRequired,          // Disc status indicates one prefix is required
  DiscStatusInvalidNumber,              // Disc status indicates invalid number
  DiscStatusInvalidAreaCode,            // Disc status indicates invalid area code
  DiscStatusNumberChanged,              // Disc status indicates number has changed
  DiscStatusRemoteBusy,                 // Disc status indicates remote line is busy
  DiscStatusNoAnswer,                   // Disc status indicates no remote answer
  DiscStatusCallRejected,               // Disc status indicates remote rejected call
  DiscStatusRemoteUnavailable,          // Disc status indicates remote is unavailable
  DiscStatusNetworkError,               // Disc status indicates network error
  DiscStatusCallPreempted,              // Disc status indicates call preempted by other call
  DiscStatusOutgoingBarred,             // Disc status indicates outgoing calls are barred
  DiscStatusIncomingBarred,             // Disc status indicates incoming calls are barred
  DiscStatusQualityUnavailable,         // Disc status indicates requested quality unavailable
  DiscStatusComputerRscUnavailable,     // Disc status indicates computer resource unavailable
  DiscStatusHWConfigurationError,       // Disc status indicates hardware configuration error
  DiscStatusChanNotIdle,                // Disc status indicates channel not idle
  DiscStatusChanTypeNotImplem,          // Disc status indicates channel type not implemented
  DiscStatusFacilityNotSubscribed,      // Disc status indicates facility not subscribed
  DiscStatusFacilityNotImplem,          // Disc status indicates facility not implemented
  DiscStatusNoRouteToDest,              // Disc status indicates no route to destination
  DiscStatusInvalidNumberFormat,        // Disc status indicates invalid number format
  DiscStatusNumberRequired,             // Disc status indicates number required
  ResultCodeEnd
} RESULTCODE;

/*-----------------------------------
 ENUMERATED CALL CONTROL CONSTANTS
 -----------------------------------*/

// INDIVIDUAL LINE STATES
typedef enum {
  LineDisconnected,                     // Line is disconnected
  LineDialed,                           // Line is dialed
  LineBusy,                             // Line is busy
  LineRing,                             // Line is ringing at local station
  LineRingback,                         // Line is ringing at remote station
  LineConnected,                        // Line is connected
  LineStateEnd
} LINESTATE;

// GENERAL CALL STATES
typedef enum {
  CallDisconnected,                     // Call is fully disconnected
  CallConnecting,                       // Call is in the process of connecting
  CallConnected,                        // Call is fully connected
  CallStateEnd
} CALLSTATE;

// CALL DESTINATION
typedef enum {
  NoDestination,                        // No specific call destination determined
  LocalStation,                         // Call to local station (incoming call)
  RemoteStation,                        // Call to remote station (outgoing call)
  LocalMCU,                             // Call to local multipoint control unit (incoming call)
  RemoteMCU,                            // Call to remote station (outgoing call)
} CALLDST;
```

```
// MULTIPOINT CALL STATE FLAGS
typedef enum {
  IsMultiConnected      = 0x0001,   // Local station is connected to more than one remote (or MCU)
  IsConfFocus           = 0x0002,   // Local station has conference focus
  IsConfChair           = 0x0004,   // Local station is conference chairman
  IsRcvingConfAudio     = 0x0008,   // Receiving conference audio
  IsRcvingConfVideo     = 0x0010,   // Receiving conference video
  IsRcvingConfData      = 0x0020,   // Receiving conference data
  IsBrdcastngAudio      = 0x0040,   // Broadcasting local audio
  IsBrdcastngVideo      = 0x0080,   // Broadcasting local video
  IsBrdcastngData       = 0x0100    // Broadcasting local data
} MULTICALLSTATE;

// CONFERENCE CONNECTIVITY PROFILE
typedef enum {
  UseAudioOnly,                     // Audio sharing only
  UseVideoOnly,                     // Video sharing only
  UseDataOnly,                      // Data sharing only
  BestDataOnly,                     // Priority to data sharing quality
  BestAudioOnly,                    // Priority to audio sharing quality
  BestVideoOnly,                    // Priority to video sharing quality
} CONFPROFILE;

// DATA TRANSFER STATES
typedef enum {
  XferReady,                        // Ready to transfer (idle)
  XferringData,                     // Transferring data
  XferRetrying,                     // Transfer retrying
  XferPaused,                       // Transfer paused
  XferFailure,                      // Transfer failed
  XferNotResponding,                // Transfer process not responding
  XferInternalError,                // Transfer process internal error
} XFERSTATE;

// MEDIA DEVICE CONTROL STATES
typedef enum {
  DeviceOpen,                       // Device is initialized and operational
  DeviceClosed,                     // Device is not operational
  DeviceFailed,                     // Device failed
  DeviceBusy,                       // Device is already in use and unavailable
  DeviceMCIFailure,                 // Device driver failure (Media Control Interface failure)
  DeviceNotResponding,              // Device is not responding
  DeviceInternalError,              // Device internal error detected
} DEVICESTATE;
```

```
/*-------------------------------------------------------------
 START CONTINUOUS LINEAR ENUMERATION OF MEDIA CONTROL OBJECT CONTROL TOKENS
-------------------------------------------------------------*/

// MEDIA CONTROL OBJECT TYPES
typedef enum {
  AudioIn = 0,
  AudioOut,                 // Audio signal from remote station
  AudioSrc,                 // Audio signal to remote station
  AudioDst,                 // Audio signal from local device
  VideoIn,                  // Audio signal to local device
  VideoOut,                 // Motion-video from remote station
  VideoSrc,                 // Motion-video to remote station
  VideoDst,                 // Motion-video from local device
  ImageIn,                  // Motion-video to local device
  ImageOut,                 // Image from remote station
  ImageSrc,                 // Image to remote station
  ImageDst,                 // Image from local device
  DataIn,                   // Image to local device
  DataOut,                  // Bit stream from remote station
  DataSrc,                  // Bit stream to remote station
  DataDst,                  // Bit stream from local device
  ObjTypeEnd                // Bit stream to local device
} MCO_TYPE;

// MEDIA CONTROL OBJECT SIGNAL TYPES
typedef enum {
  SignalIn = ObjTypeEnd,
  SignalOut,                // signal from remote station
  SignalSrc,                // signal to remote station
  SignalDst,                // signal from local media control device
  SignalTypeEnd             // signal to local media control device
} MCO_SIGTYPE;

// MEDIA CONTROL OBJECT COMPOSITE TYPE
typedef enum {
  Discreet = SignalTypeEnd,
  Merged,                   // Multiple inputs to same multiple outputs
  Multiplexed,              // Multiple inputs mixed into complex single output
  Demultiplexed,            // Multiple inputs encoded into single output
  Transformed,              // single input decoded into multiple outputs
  CompositeTypeEnd          // single input subjected to specific transform
} MCO_COMPTYPE;

// DATA TRANSFER OBJECTS
typedef enum {
  XferNoObject = CompositeTypeEnd,
  XferCursorPos,            // No specified data transfer object
  XferString,               // Cursor Position
  XferTextFile,             // Null-terminate ASCII text string
  XferBinFile,              // Text file
  XferObjEnd                // Binary file
} MCO_XFEROBJ;
```

```
// MEDIA CONTROL OBJECT SETTINGS
typedef enum {

    // BASE OBJECT SETTINGS
3960    NoSetting = XferObjEnd,          // No specific setting
        Open,                            // Open object (initialize and render operational)
        Close,                           // Close object
        Enable,                          // Enable object (make available for use)
        Disable,                         // Disable object (make unavailable for use)
3965    On,                              // Turn on object signal
        Off,                             // Turn off object signal
        AttachTo,                        // Attach object signal to another object signal
        DetachFrom,                      // Detach object signal from another object signal
        AddToComposite,                  // Add object signal to composite signal
3970    RemoveFromComposite,             // Remove object signal to composite signal
        SetCompositeType,                // Set modality of composite signal
        GetStatus,                       // Get status of object signal
        GetCaps,                         // Get capabilities for object

3975    // MOTION-VIDEO SETTINGS
        SetColorkey,                     // Set motion-video color-key value for display
        AssignWindow,                    // Assign motion-video display to specified window
        UnassignWindow,                  // Unassign motion-video display from window
        ResizeWindow,                    // Resize (refresh and realign)motion-video window
3980    SetStretchOn,                    // Set motion-video stretch mode on
        SetStretchOff,                   // Set motion-video stretch mode off
        SetImageType,                    // Set motion-video image type
        Freeze,                          // Freeze motion-video signal
        Unfreeze,                        // Unfreeze motion-video signal
3985    SetProportionalOn,               // Set motion-video proportional mode on
        SetProportionalOff,              // Set motion-video proportional mode off
        SetVideoFrameSize,               // Set video frame size

        // IMAGE SETTINGS
3990    /* AssignWindow,                 // Assign imaging display to window (already defined above)
        UnassignWindow,                  // Unassign imaging display from window (already defined above)
        ResizeWindow,                    // Resize (refresh/realign) imaging window (already defined above)
        SetStretchOn,                    // Set imaging stretch mode on (already defined above)
        SetStretchOff,                   // Set imaging stretch mode off (already defined above)
3995    SetImageType, */                 // Set imaging image type (already defined above)
        SetImageMetric,                  // Set imaging image metric type
        SetPixelWidth,                   // Set imaging image pixel width
        SetPixelHeight,                  // Set imaging image pixel height
        SetPixelDepth,                   // Set imaging image pixel depth
4000    SetPhysicalWidth,                // Set imaging image physical width
        SetPhysicalHeight,               // Set imaging image physical height
        SetHorzPixelOrigin,              // Set horizontal image pixel origin
        SetVertPixelOrigin,              // Set vertical image pixel origin
        SetHorzPhysicalOrigin,           // Set horizontal image physical origin
4005    SetVertPhysicalOrigin,           // Set vertical image pixel origin
        SetHorzPixelDensity,             // Set horizontal image pixel density
        SetVertPixelDensity,             // Set vertical image pixel density
        SetImageCombineType,             // Set image combine type

4010    // AUDIO SETTINGS
        SetAudioQuality,                 // Set audio signal quality
        LipSynchOn,                      // Turn on lip-synchronization of audio signal to video signal
        LipSynchOff,                     // Turn off lip-synchronization of audio signal to video signal
        EchoCancelOn,                    // Turn echo cancellation on
4015    EchoCancelOff,                   // Turn echo cancellation off
        SetDTMFDuration,                 // Set dial tone modulation frequency pulse duration
        LocalDTMFPulse,                  // Pulse DTMF at local station
        RemoteDTMFPulse,                 // Pulse DTMF at remote station

4020    // DATA SETTINGS
        SetDataRate,                     // Set data transfer rate
        SetSyncXferMode,                 // Set synchronous data transfer mode
```

```
    SetAsyncXferMode,                   // Set asynchronous data transfer mode
    SetRestrictedMode,                  // Set restricted data transfer mode
4025 SetUnrestrictedMode,               // Set unrestricted data transfer mode
    McoSettingEnd
  } MCO_SETTING;

  // BASIC IMAGE TYPES
4030 typedef enum {
    NoImage = McoSettingEnd,            // No image available
    ColorImage,                         // Color image type
    GrayscaleImage,                     // Grayscale image type
    BitonalImage,                       // Two-tone image type
4035 ImageTypeEnd
  } IMAGETYPE;

  // IMAGE METRICS
  typedef enum {
4040 InchMetrics = ImageTypeEnd,        // Set "inch" as primary measure
    CentiMetrics,                       // Set "centimeter" as primary measure
    MilliMetrics,                       // Set "millimeter" as primary measure
    MicroMetrics,                       // Set "micrometer" as primary measure
    ImageMetricEnd
4045 } IMAGEMETRIC;

  // IMAGE-ON-IMAGE COMBINE TYPES
  typedef enum {
    Overlay = ImageMetricEnd,           // Overlay destination with source
4050 Replace,                           // Replace destination with source
    ColorKey,                           // Overlay destination defined by colorkey with source
    OutLine,                            // Overlay destination with temporary outline of source
    BitwiseOR,                          // Combine destination and source with bitwise OR
    BitwiseXOR,                         // Combine destination and source with bitwise XOR
4055 BitwiseAND,                        // Combine destination and source with bitwise AND
    ImageCombineTypeEnd
  } IMAGECOMBTYPE;

  // MOTION-VIDEO FRAME SIZES (ITU-T H.261)
4060 typedef enum {
    NoVideo = ImageCombineTypeEnd,      // No video signal
    QuarterCIF,                         // Quarter-size Common Intermediate Format video image
    FullCIF,                            // Full-size Common Intermediate Format video image
    CIF240,                             // Common Intermediate Format video image with 240 scanlines
4065 4CIF,                              // Four-times Common Intermediate Format video image
    VideoSizeEnd
  } VIDEOSIZE;

  // AUDIO SIGNAL QUALITY
4070 typedef enum {
    NoAudio = VideoSizeEnd,             // No audio signal
    VoiceLow,                           // Low quality voice signal (usually 8khz sample rate)
    VoiceHigh,                          // High quality voice signal (usually 8-11khz sample rate)
    Music,                              // Music quality signal (usually 22khz sample rate)
4075 HighFidelity,                      // High fidelity quality signal (usually 44khz sample rate)
    AudioQualityEnd
  } AUDIOQUALITY;
```

```
// DATA TRANSFER RATES
typedef enum {
  DataRateNone = AudioQualityEnd,          // No data transfer
  DataRate300,                             // 300 baud transfer rate
  DataRate1200,                            // 1200 baud transfer rate
  DataRate4800,                            // 4800 baud transfer rate
  DataRate9600,                            // 9600 baud transfer rate
  DataRate14Kb,                            // 14.4 kilobaud transfer rate
  DataRate28Kb,                            // 28.8 kilobaud transfer rate
  DataRate64Kb,                            // 64 kilobaud transfer rate
  DataRate128Kb,                           // 128 kilobaud transfer rate
  DataRate192Kb,                           // 192 kilobaud transfer rate
  DataRate256Kb,                           // 256 kilobaud transfer rate
  DataRate320Kb,                           // 320 kilobaud transfer rate
  DataRate384Kb,                           // 384 kilobaud transfer rate
  DataRate512Kb,                           // 512 kilobaud transfer rate
  DataRate1152Kb,                          // 1152 kilobaud transfer rate
  DataRate1536Kb,                          // 1536 kilobaud transfer rate
  DataRateEnd
} DATARATE;

// LAST VALID MCO TOKEN VALUE (USED FOR BOUNDS CHECKING OF ARGUMENTS)
typedef enum {
  MediaControlTokenEnd = DataRateEnd
};

/*-------------------------------------------------------
END CONTINUOUS LINEAR ENUMERATION OF MEDIA CONTROL OBJECT CONTROL TOKENS
-------------------------------------------------------*/

// STRUCTURE FOR VCO EVENT DESCRIPTOR
struct tagEVENTREC {
  DWORD       Id;                          // 32-bit VCO event identifier
  DWORD       Param1;                      // 32-bit event parameter 1
  DWORD       Param2;                      // 32-bit event parameter 2
  STATION*    pStation;                    // Ptr to source station
  BOOL        IsFromDevice;                // True if event generated by encapsulated device
  tagEVENTREC* pNext;                      // Ptr to next event in queue or list
  tagEVENTREC* pPrev;                      // Ptr to previous event in queue or list
};

typedef tagEVENTREC EVENTREC;

// STRUCTURE FOR STATION DESCRIPTOR
struct tagSTATION {
  DWORD       Id;                          // System identifier/index used to refer to this station
  char*       pLabel;                      // Ptr to station label
  char*       pNumber[3];                  // Array of ptrs to numbers of remote station
  BOOL        IsVco;                       // True if remote station is determined to be a VCO
  tagSTATION* pNext;                       // Ptr to next station in list
  tagSTATION* pPrev;                       // Ptr to previous station in list
};

typedef tagSTATION STATION;

// DEFINITION OF EVENT HANDLING MEMBER FUNCTION
typedef DWORD EVENTPROC(
  EVENT       _Id,                         // 32-bit event identifier
  DWORD       _Param1,                     // 32-bit event parameter 1
  DWORD       _Param2,                     // 32-bit event parameter 2
  STATION*    _pStation,                   // Ptr to descriptor for station originating event
  HNOTIFIER*  _hNotifier                   // Handle to notification object triggered by event
);
```

```
// STRUCTURE FOR VCO NOTIFIER DESCRIPTOR
typedef struct {
  DWORD       Triggers;          // Mask specifying events that trigger this notifier
  void*       pObject;           // Ptr to Notifier Receiver Object (NRO)
4145 EVENTPROC*  pMember;         // Ptr to notifier handler member of NRO
  BOOL        IsEnabled;         // True if notifier is enabled to trigger
  BOOL        OnlyDeviceEvents;  // True if notifier triggers only for device events
  long        nTriggered;        // Number of times notifier triggered since last reset
  DWORD       ReturnData;        // Data returned by notification handler member of NRO
4150 } NOTIFIER;

// STRUCTURE FOR RED-GREEN-BLUE COLOR SPECIFICATION
typedef struct {
  BYTE        Red;
4155 BYTE        Green;             // Red color component
  BYTE        Blue;              // Green color component
  BYTE        reserved;          // Blue color component
} RGBVALUE;

4160 // STRUCTURE FOR DEVICE DESCRIPTOR
typedef struct {
  DEVICESTATE State;             // State of physical device
  char*       pLabel;            // Ptr to label for physical device
  char*       pVersion;          // Ptr to version string for physical device
4165 int         nObjects;          // Number of media ctrl objects associated with physical device
  HMCO        phMCO;             // Ptr to array of handles for media ctrl objs associated with device
} DEVICE;

// STRUCTURE FOR MEDIA CONTROL OBJECT AUDIO PARAMETERS
4170 typedef struct {
  AUDIOQUALITY Quality;          // MCO audio quality
  BOOL        IsLipSynched;      // True if audio lip-synchronized with video signal
  HMCO        hVideoObj;         // Handle to lip-synchronized video object
  BOOL        IsEchoCancelOn;    // True if echo cancellation is enabled
4175 int         DTMFDuration;      // Dial Tone Modulation Frequency duration in msec
} MCO_AUDIOPARAM;

// STRUCTURE FOR MEDIA CONTROL OBJECT MOTION-VIDEO PARAMETERS
typedef struct {
4180 BOOL        IsAssignedToWin;   // True if obj assigned to window
  BOOL        IsWinUpdated;      // True if window aligned with source video image
  BOOL        IsFrozen;          // True if motion-video is frozen
  BOOL        IsProportional;    // True if motion-video proportional mode is on
  BOOL        IsStretched;       // True if motion-video stretch mode is on
4185 IMAGETYPE   ImageType;         // Motion-video image type
  VIDEOSIZE   VideoSize;         // Motion-video frame size
  Window*     pDisplayWin;       // Ptr to assigned display window
} MCO_VIDEOPARAM;
```

```
4190    // STRUCTURE FOR MEDIA CONTROL OBJECT IMAGING PARAMETERS
        typedef struct {
          BOOL            IsAssignedToWin;    // True if object assigned to window
          BOOL            IsWinUpdated;       // True if window aligned with source video image
          BOOL            IsFrozen;           // True if imaging is frozen
4195      BOOL            IsProportional;     // True if imaging proportional mode is on
          BOOL            IsStretched;        // True if imaging stretch mode is on
          IMAGETYPE       BasicType;          // Basic image type
          IMAGECOMBTYPE   CombType;           // Image combine type
          IMAGEMETRIC     ImageMetric;        // Image primary measure
4200      int             PixelWidth;         // Image pixel width
          int             PixelHeight;        // Image pixel height
          int             PixelDepth;         // Image pixel depth
          int             HorzPixelOrigin;    // Image horizontal pixel origin
          int             VertPixelOrigin;    // Image vertical pixel origin
4205      int             HorzPixelDensity;   // Image horizontal pixel density
          int             VertPixelDensity;   // Image vertical pixel density
          int             PhysicalWidth;      // Image physical width
          int             PhysicalHeight;     // Image physical height
          int             HorzPhysicalOrigin; // Image horizontal physical origin
4210      int             VertPhysicalOrigin; // Image vertical physical origin
          Window*         pDisplayWin;        // Ptr to assigned display window
        } MCO_IMAGEPARAM;

        // STRUCTURE FOR MEDIA CONTROL OBJECT DATA PARAMETERS
4215    typedef struct {
          BOOL            IsSynchronous;      // True if data transfer is synchronous
          BOOL            IsRestricted;       // True if bandwidth is restricted
          BOOL            IsComposite;        // True if part of composite
          DATARATE        TransferRate;       // Data transfer rate
4220      int             CompositeRate;      // Composite transfer rate (if part of composite)
        } MCO_DATAPARAM;

        // STRUCTURE FOR MEDIA CONTROL OBJECT DESCRIPTOR
        typedef struct tagMCO {
4225      char*           pLabel;             // Ptr to label for media ctrl object
          MCO_TYPE        ObjType;            // Media ctrl object type
          MCO_SIGTYPE     SigType;            // Media ctrl object signal type
          BOOL            IsValid;            // True if media ctrl object is valid service or place holder
          BOOL            IsOpen;             // True if media ctrl object open
4230      BOOL            IsEnabled;          // True if media ctrl object is enabled
          BOOL            IsOn;               // True if media ctrl object is on
          BOOL            IsAttached;         // True if media ctrl object is attached to another media ctrl object
          BOOL            IsComposite;        // True if media ctrl object is part of composite
          BOOL            IsBusy;             // True if media ctrl object is busy (unavailable)
4235      BOOL            IsEncoded;          // True if signal is encode or compressed; false if not
          MCO_AUDIOPARAM  Audio;              // Audio settings parameter block (if audio type)
          MCO_VIDEOPARAM  Video;              // Video parameter block (if video type)
          MCO_IMAGEPARAM  Image;              // Image parameter block (if image type)
          MCO_DATAPARAM   Data;               // Data parameter block (if data type)
4240      DEVICE*         pDevice;            // Ptr to struct for device with which media ctrl object is associated
        } MCOPARAM;

        // STRUCTURE FOR MEDIA CONTROL OBJECT BINDING RECORD
        struct tagMCO_BINDING {
4245      BOOL            IsComposite;        // True if binding is to produce composite signal
          int             nSrc;               // Number of source media ctrl objects
          int             nDst;               // Number of destination media ctrl objects
          HMCO            phMcoSrc;           // Ptr to list of handles for source media ctrl objects
          HMCO            phMcoDst;           // Ptr to list of handles for destination media ctrl object
4250      tagMCO_BINDING* pNext;              // Ptr to next binding record
          tagMCO_BINDING* pPrev;              // Ptr to prev binding record
        };

        typedef tagMCO_BINDING MCO_BINDING;
```

```
4255  // STRUCTURE FOR MEDIA CONTROL OBJECT COMMAND RECORD
      typedef struct {
        MCO_TYPE        Type;       // Target media ctrl object type
        MCO_SETTING     Setting;    // Setting for media ctrl object
        DWORD           Param;      // Parameter for setting
4260  } MCO_CMD;

      // STRUCTURE FOR MODE-CAPABILITY CROSS-REFERENCE RECORD (ITU-T H.221)
      typedef struct {
        DWORD           Value;      // Mode or capability value to be cross-referenced
4265    int             nRefs;      // Number of cross-references for mode or cap
        DWORD           Ref[MaxXRef]; // List of referenced modes or caps
      } XREF;

      // STRUCTURE FOR DEVICE CAPABILITIES LISTING (ITU-T H.221)
4270  typedef struct {
        int
        BASCODE         nCaps;      // Number of H.221 device capabilities
      } DEVCAPS;                    Cap[MaxCaps]; // Listing of H.221 device capabilities

4275  // STRUCTURE FOR CAPABILITIES DATA (ITU-T H.221)
      typedef struct {
        DEVCAPS         Local;      // Local device capabilities listing
        DEVCAPS         Remote;     // Remote device capabilities listing
        DEVCAPS         Connect;    // Connectivity (network interface) capabilities listing
4280    int             nModes;     // Number of entries in "Modes to Caps" xref list
        int             nCaps;      // Number of entries in "Caps to Modes" xref list
        XREF            Caps[MaxCaps]; // "Caps to Modes" xref list
        XREF            Modes[MaxModes]; // "Modes to Caps" xref list
      } CAPS;
4285
      // STRUCTURE FOR MEDIA CONTROL DEVICE PARAMETERS
      typedef struct {
        const int       Devices;    // Number of encapsulated devices
        const DEVICE    Dev[MaxDevices]; // Encapsulated device chain
4290    CAPS            Cap;        // H.221 capabilities for VCO devices
        int             nMco;       // Number of media ctrl objects currently available
        int             nAudioObj;  // Number of audio objects currently available
        int             nVideoObj;  // Number of motion-video objects
        int             nImageObj;  // Number of image objects
4295    int             nDataObj;   // Number of data objects
        const char*     pMcoLabel[]; // Ptr to array of ptrs to media ctrl object labels
        MCO_BINDING     pMcoBinding; // Ptr to linked list of current media ctrl object bindings
        HMCO*           phMco;      // Ptr to list of handles to all available media ctrl objects
        HMCO            hMco[MaxMcoType]; // Default media ctrl object handles (reference with type enum)
4300  } DEVICEPARAM;

      // STRUCTURE FOR CONFIGURATION AND SETUP PARAMETERS
      typedef struct {
        BOOL            IsDynaPortable; // True if VCO is dynamically re-loadable at run-time
4305    BOOL            IsMultiConnectable; // True if VCO supports multipoint control operations
        BOOL            IsMultiInstantiable; // True if VCO supports multiple concurrent instances
        BOOL            IsRestricted; // True if service bandwidth is restricted
        BOOL            IsEmulating; // True if VCO starts up emulating devices
        STATION         LocalStation; // Label and numbers for local station
4310    STATION         RemoteStation; // Label and numbers for default remote station
        char*           TermOutputDevice; // Default terminal output device or file name
        int             ConnectTimeout; // Default connection timeout in msec.
        int             DeviceTimeout; // Default device timeout in msec.
        int             DispatcherRate; // Starting dispatcher rate in msec.
4315    int             ServiceBandwidth; // Total service bandwidth available
        int             nLinesAvailable; // Number of lines available
        int             nLinesRequested; // Number of lines request for use by this VCO
        RGBVALUE        ColorKey;   // colorkey value for motion-video
        CONFPROFILE     ConfProfile; // Conference profile
4320  } CONFIGPARAM;
```

```
// STRUCTURE FOR CALL CONTROL PARAMETERS FOR CURRENT CALL
typedef struct {
    CALLSTATE       State;          // Call state for entire call
    CONFPROFILE     ConfProfile;    // This conference profile
    CALLDST         CallDst;        // Destination for call
    RESULTCODE      DiscStatus;     // Disconnect status (when in disconnected state)
    int             nLines;         // Total number of lines to be used for this call
    BOOL            IsRestricted;   // True if this call is restricted
    BOOL            IsCallSetup;    // True if this call is setting up
    BOOL            IsCallingVco;   // True if this call destination is another VCO
    int             nConnections;   // Number of current connections for this call
    int             Bandwidth;      // Total bandwidth used for this call
    int             Timeout;        // Call connect timeout used for this call
    int             TimeSlots;      // Timeslots used for this call (if applicable)
    LINESTATE       LineState[3];   // Linestate for each line in call
    int             nStations;      // Total number of stations involved in conference
    STATION         pStations;      // Ptr to list of conference stations (first is local)

// MULTIPOINT CALL CONTROL PARAMETERS FOR CURRENT CALL
    MULTICALLSTATE  MultiCallStates; // Multipoint call status flags
    BOOL            IsConfFocus;    // True if station has conference focus
    BOOL            IsConfChair;    // True if station is conference chairman
    BOOL            IsRcvingConfAudio; // True if station is receiving conference audio
    BOOL            IsRcvingConfVideo; // True if station is receiving conference video
    BOOL            IsRcvingConfData; // True if station is receiving conference data
    BOOL            IsBrdcastngAudio; // True if station is broadcasting audio
    BOOL            IsBrdcastngVideo; // True if station is broadcasting video
    BOOL            IsBrdcastngData; // True if station is broadcasting data
} CALLPARAM;

// STRUCTURE FOR CONNECTIVITY PROTOCOL PARAMETERS (ITU-T H.320, ITU-T H.221)
typedef struct {
    BOOL            IsXmtngAudio;   // True if transmitting audio
    BOOL            IsXmtngVideo;   // True if transmitting video
    BOOL            IsXmtngData;    // True if transmitting data
    BOOL            IsRcvingAudio;  // True if receiving audio
    BOOL            IsRcvingVideo;  // True if receiving video
    BOOL            IsRcvingData;   // True if receiving data
    BASCODE         RcvDataRate;    // Current receive transfer rate
    BASCODE         RcvAudioMode;   // Current receive audio mode
    BASCODE         RcvVideoMode;   // Current receive video mode
    BASCODE         RcvDataMode;    // Current receive data mode
    BASCODE         XmtDataRate;    // Current transmit transfer rate
    BASCODE         XmtAudioMode;   // Current transmit audio mode
    BASCODE         XmtVideoMode;   // Current transmit video mode
    BASCODE         XmtDataMode;    // Current transmit data mode
    BASCODE         NewDataRate;    // Pending transfer rate just set (pending XmtDataRate)
    BASCODE         NewAudioMode;   // Pending audio mode just set (pending XmtAudioMode)
    BASCODE         NewVideoMode;   // Pending video mode just set (pending XmtVideoMode)
    BASCODE         NewDataMode;    // Pending data mode just set (pending XmtDataMode)
    int             nMiscXmtMode;   // Number of miscellaneous modes set by local station
    int             nMiscRcvMode;   // Number of miscellaneous modes set by remote station
    BASCODE         MiscXmtMode[MaxMiscMode]; // List of miscellaneous modes set by local station
    BASCODE         MiscRcvMode[MaxMiscMode]; // List of miscellaneous modes set by local station
} PROTOCOLPARAM;

// STRUCTURE FOR REMOTE STATION CONTROL PARAMETERS
typedef struct {
    BOOL            IsAttached;     // True if cmd and event stream attached to remote VCO
    BOOL            IsMaster;       // True if controlling remote station (master)
    BOOL            IsSlave;        // True if controlled by remote station (slave)
    CONTROLMODE     Modes;          // Control mode setting flags
    CONTROLMODE     Caps;           // Control mode capability/permission flags
} CONTROLPARAM;
```

```
// STRUCTURE FOR REMOTE STATION MONITORING PARAMETERS
typedef struct {
    BOOL        IsAttached;     // True if event stream attached to remote VCO
    BOOL        IsMonitoring;   // True if monitoring at least one remote station
4390 BOOL       IsMonitored;    // True if monitored by remote station
    int         nStation;       // Number of stations currently monitored
    STATION*    pStations;      // Ptr to list of stations currently monitored
    MONITORMODE Modes;          // Monitor mode setting flags
    MONITORMODE Caps;           // Monitor mode capability/permission flags
4395 } MONITORPARAM;

// STRUCTURE FOR VCO SYSTEM INFORMATION (VCO PARAMETER BLOCK)
typedef struct {
    char*       pLabel;         // Ptr to VCO label string
4400 char*      pVersion;       // Ptr to VCO version string
    VCOSTATE    State;          // VCO global operational state
    int         RefCount;       // VCO reference count of users
    BOOL        IsEmulating;    // True if emulating devices
    BOOL        IsReady;        // True if ready to connect to remote station
4405 DEVICEPARAM Device;        // VCO encapsulated device parameter block
    CONFIGPARAM Config;         // VCO configuration parameter block
    CALLPARAM   Call;           // VCO current call parameter block
    PROTOCOLPARAM Protocol;     // VCO protocol parameter block
    CONTROLPARAM Control;       // VCO control context parameter block
4410 MONITORPARAM Monitor;      // VCO monitoring context parameter block
} VCOPARAM;
```

```
/*-----------------------------------------------------------------
                              CLASS VDI
                        (Virtual Device Interface)

Below is the declaration for the VCO Virtual Device Interface Class. An instance of the VDI class must contain, at a minimum, all
of the public member functions use by VCO clients to establish and control multimedia connectivity sessions with remote stations.
This class must also contain an instance of a VCOPARAM data structure, and the pure virtual member declarations for the device
control member functions that provide device support to the public member function implementations. The implementations for these
pure virtual functions (demarked with the Dev<label> symbolic naming convention) reside in the Physical Device Interface (class
PDI). The constructor and destructor of this class, are protected members, and their public interface is via call from
constructor/destructor in the more derived class VCO.
-----------------------------------------------------------------*/

class VDI: protected EVENT {
    protected:

        // MULTIMEDIA CONNECTION SYSTEM INFORMATION
        VCOPARAM VcoParam;

        // INTERNAL DEVICE-INDEPENDENT MEMBERS GO HERE...
            .
            .
            .

        virtual const char* GetClassName() { return "VDI"; };

        /*-------------------------
         NETWORK SESSION CONTROL
         -------------------------*/

        VDI( char* _pInitFile = 0 );
        /*
            USAGE:  Construct the Virtual Device Interface for the VCO. Initialize VCO parameters and
                    settings from the specified initialization file. Setup device-independent data and code
                    objects used by VCO. Create the default VCO device event notifier and start the VCO
                    dispatcher.

            PARAM:  _pInitFile          ...Filespec of file that contains VCO startup params & settings.

            RETURN: none
        */

        virtual ~VDI();
        /*
            USAGE:  Destruct the Virtual Device Interface. Save current settings to the initialization file. Close
                    the various media ctrl objects, if open. Delete any notifiers and stop the VCO dispatcher.
                    Free all resources allocated by VCO.

            PARAM:  none

            RETURN: none
        */
    public:
```

```
RESULTCODE Open( BOOL _IsBlocking = 1 );
/*
    USAGE:  Prepare the VCO for making call to remote station. Initialize all media ctrl ojects
            and their supporting device control sub-systems. Perform preliminary sub-system
            diagnostics and determine level of system functionality.

    PARAM:  _IsBlocking          ...True if call is blocking & will not return until complete, or false if
                                 non-blocking & returns immediately as "pending".

    RETURN: Failure
            Success
            Pending
            TimedOut
            Redundant
            Disabled
            MemoryAllocError
            ResourceAllocError
            InternalError
            TimerFailure
*/

RESULTCODE Close( BOOL _IsBlocking = 1 );
/*
    USAGE:  Shutdown the VCO. Stop all services provided by media ctrl objects and close
            their supporting device control sub-systems. Free resources allocated for device control.

    PARAM:  _IsBlocking          ...True if call is blocking & will not return until complete, or false if
                                 non-blocking & returns immediately as "pending".

    RETURN: Failure
            Success
            Pending
            TimedOut
            MustBeOpened
            Disabled
            MemoryAllocError
            ResourceAllocError
            InternalError
*/
```

```
RESULTCODE Call( char*   _pNumber1 = 0,
                 char*   _pNumber2 = 0,
                 BOOL    _IsBlocking = 1 );
/*
    USAGE:  Establish mutual call to remote station, or MCU; create connectivity session whose quality is
            determined by the highest common denominator of media curl connectivity services,
            as may be accommodated by both local and remote stations. A preference as to the quality of
            this interaction is expressed by each station; subsequently proceeds negotiation, between
            these stations, to establish the most appropriate media device interconnection modalities
            requisite to best fulfilling the requests for specific (at times conflicting) conference profiles.

    PARAM:  _pNumber1       ...Ptr to string with number for line 1, null calls default remote station.

            _pNumber2       ...Ptr to string with number for line 2 (if used).

            _IsBlocking     ...True if call is blocking & will not return until complete, or false if
                               non-blocking & returns immediately as "pending".
    RETURN: Failure
            Success
            Pending
            TimedOut
            MustBeOpened
            Disabled
            InUse
            MemoryAllocError
            ResourceAllocError
            InternalError
            TimerFailure
            InvalidDataType
            NotEnoughBandwidth
*/
```

```
RESULTCODE MultiCall( STATION*      _pStation,
                      MULTICALLOP   _Op,
                      DWORD         _Param = 0,
                      BOOL          _IsQuery = 0,
                      BOOL          _IsBlocking = 1 );
/*
    USAGE.  Establish multipoint call by presenting a multipoint control operation request to the
            connectivity sub-system, while currently connected to multipoint control unit (MCU). This
            function allows a station to participate in a conference with more than two conferees, to
            control a conference, and to direct local/common media ctrl to/from conferees.

    PARAM:  _Station        ...Ptr to station descriptor specifying to which station operation applies.

            _Op             ...Multipoint call control operation specifier.

            _Param          ...Parameter for specified operation.

            _IsQuery        ...True if call is to query sub-system for operation capability.

            _IsBlocking     ...True if call is blocking & will not return until complete, or false if
                               non-blocking & returns immediately as "pending".

            MULTIPOINT CALL CONTROL OPERATION USAGE & PARAMETERS

            < _Op >                 < _Param >

            GetNumStations          ..Ptr to int to receive count of stations in conf
            GetStationList          ...Ptr to buffer to hold linked list of STATION records
            GetStationCaps          ...Ptr to DEVCAPS record
            GetStationIdentity      ...Ptr to STATION record to rcv id of remote station
            ...all other ops        ...Don't care

    RETURN: Failure
            Success
            Pending
            TimedOut
            Redundant
            RequestDenied
            NotSupported
            MustBeOpened
            Disabled
            InUse
            InternalError
            InvalidStation
            InvalidDataType
            InvalidOperation
            InvalidOperationNow
            InvalidParam
            CallMustBeConnected
            NoCallForLineAdd
*/
```

```
                RESULTCODE Hangup( int _nLine = 0 );
4590            /*
                    USAGE:  Hang-up entire call to remote station, or MCU; selectively disconnect specified line only.

                    PARAM:  _nLine                  ...Number of lines to disconnect; null hangs up all lines.

4595                RETURN: Failure
                            Success
                            TimedOut
                            MustBeOpened
                            Disabled
4600                        InternalError
                            InvalidLine
                            CallMustBeConnected
                            LineIsDown
                            LineNotConnected
4605            */
```

```
/*-----------------------------
 EVENT NOTIFICATION CONTROL
 -----------------------------*/

RESULTCODE NewNotifier( HNOTIFIER&    _hNotifier,
                        EVENTPROC*    _pMember,
                        void*         _pObject,
                        DWORD         _EventMask = 0 );
/*
    USAGE:  Create new notification object in the VCO linked noufication object list. The noufier is
            initially "disabled" following creation, and must be enabled to trigger.

    PARAM:  _hNotifier      ...Reference to handle for newly created VCO noufication object.

            _pMember        ...Ptr to noufier receiver member to process VCO events.

            _pObject        ...Ptr to noufication receiver object.

            _EventMask      ...Mask specifying events that will trigger noufier.

    RETURN: Failure
            Success
            RequestDenied
            Disabled
            InvalidDataType
            InvalidParam
            MemoryAllocError
            InternalError
*/

RESULTCODE DeleteNotifier( HNOTIFIER _hNotifier );
/*
    USAGE:  Delete VCO signal and remove it from VCO linked object list.

    PARAM:  _hNotifier      ...Handle to signal to be deleted.

    RETURN: Failure
            Success
            RequestDenied
            Disabled
            InvalidDataType
            InvalidNotifier
*/

RESULTCODE EnableNotifier( HNOTIFIER _hNotifier,
                           BOOL      _IsEnabled = 1 );
/*
    USAGE:  Enable or disable signal from triggering on its specified triggering events.

    PARAM:  _hSignal        ...Handle to signal to be enabled or disabled.

            _IsEnabled      ...True enables signal triggering; false disables triggering.

    RETURN: Failure
            Success
            Redundant
            Disabled
            InvalidDataType
            InvalidNotifier
*/
```

```
RESULTCODE SetNotifierTriggers( HNOTIFIER  _hNotifier,
                                DWORD      _EventMask = 0 );
/*
    USAGE:  Set events that will trigger signal

    PARAM:  _hNotifier          ...Handle to signal whose trigger events will be set.

            _EventMask          ...Mask specifying events that will trigger signal.

    RETURN: Failure
            Success
            Disabled
            InvalidDataType
            InvalidNotifier
*/

RESULTCODE TriggerNotifiers( EVENTREC*   _pEventRec,
                             HNOTIFIER   _hNotifier = 0 );
/*
    USAGE:  Triggers VCO notifiers sensitive to the specified event, or alternatively trigger a specific signal.

    PARAM:  _pEventRec          ...Ptr to record containing event parameters.

            _hNotifier          ...If specified, indicates specific signal to be triggered with event, or
                                else  all notifiers sensitive to event are triggered.

    RETURN: Failure
            Success
            Disabled
            InvalidDataType
            InvalidNotifier
            InvalidParam
*/
```

```
/*--------------------
 CONFIGURATION/SETUP CONTROL
 --------------------*/

RESULTCODE SetConfig( CONFIGPARAM* _pConfig );
/*
    USAGE:  Set VCO configuration data for main object and encapsulated devices.

    PARAM:  _pConfig            ...Ptr to record containing new configuration

    RETURN: Failure
            Success
            RequestDenied
            Disabled
            InvalidDataType
            InvalidParam
            InternalError
            TimerFailure
*/

RESULTCODE StoreConfig( CONFIGPARAM* _pConfig = 0 );
/*
    USAGE:  Store VCO configuration to backing store.

    PARAM:  _pConfig            ...Ptr to record containing configuration to write to backing store. If
                                none specified, the current VCO config is stored.

    RETURN: Failure
            Success
            RequestDenied
            Disabled
            InvalidDataType
            InvalidParam
            InternalError
            TimerFailure
*/

RESULTCODE RefreshConfig( CONFIGPARAM* _pConfig = 0 );
/*
    USAGE:  Refreshes current VCO configuration or configuration record from that saved in backing store.

    PARAM:  _pConfig            ...Ptr to record to receive configuration read from backing store. If none
                                specified, current VCO config is refreshed.

    RETURN: Failure
            Success
            RequestDenied
            Disabled
            InvalidDataType
            InvalidParam
            InternalError
            TimerFailure
*/
```

```
RESULTCODE SetupAudioDevices( BOOL _IsBlocking = 1 );
/*
    USAGE:  Invokes dialog box to enable interactive setup of audio devices.

    PARAM:  _IsBlocking          ...True if call is blocking & will not return until complete; false if
                                 non-blocking & returns immediately as "pending".

    RETURN: Failure
            Success
            Pending
            Redundant
            RequestDenied
            NotSupported
            ProcessTerminated
            MustBeOpened
            Disabled
            InUse
            MemoryAllocError
            ResourceAllocError
            InternalError
*/

RESULTCODE SetupVideoDevices( BOOL _IsBlocking = 1 );
/*
    USAGE:  Invokes dialog box to enable interactive setup of motion-video devices.

    PARAM:  _IsBlocking          ...True if call is blocking & will not return until complete; false if
                                 non-blocking & returns immediately as "pending".

    RETURN: Failure
            Success
            Pending
            Redundant
            RequestDenied
            NotSupported
            ProcessTerminated
            MustBeOpened
            Disabled
            InUse
            MemoryAllocError
            ResourceAllocError
            InternalError
*/

RESULTCODE SetupImageDevices( BOOL _IsBlocking = 1 );
/*
    USAGE:  Invokes dialog box to enable interactive setup of image devices.

    PARAM:  _IsBlocking          ...True if call is blocking & will not return until complete; false if
                                 non-blocking & returns immediately as "pending".

    RETURN: Failure
            Success
            Pending
            Redundant
            RequestDenied
            NotSupported
            ProcessTerminated
            MustBeOpened
            Disabled
            InUse
            MemoryAllocError
            ResourceAllocError
            InternalError
*/
```

```
RESULTCODE SetupDataDevices( BOOL _IsBlocking = 1 );
/*
    USAGE:  Invokes dialog box to enable interactive setup of data connectivity and network adapter
            devices (Network Interface Units), as well as allow configuration of system I/O ports. Setup
            of network protocol support software resides here.

    PARAM:  _IsBlocking          ...True if call is blocking & will not return until complete, or false if
                                 non-blocking & returns immediately as "pending".

    RETURN: Failure
            Success
            Pending
            Redundant
            RequestDenied
            NotSupported
            ProcessTerminated
            MustBeOpened
            Disabled
            InUse
            MemoryAllocError
            ResourceAllocError
            InternalError
*/
```

```
/*----------
 MEDIA CONTROL
 ----------*/

RESULTCODE MediaControl( MCO_TYPE      _McoType,
                         MCO_SETTING   _Setting,
                         DWORD         _Param = 0,
                         BOOL          _IsQuery = 0,
                         BOOL          _IsBlocking = 1 );
/*
    USAGE:  Access service provided by encapsulated media control device by presenting media ctrl
            control setting to physical device control sub-system.

    PARAM:  _McoType        ...Specific media ctrl object type for operation.

            _Setting        ...Audio-video-data setting constant specifying requested service
                               desired from object.

            _Param          ...If required, provides parameter necessary to fully specify request to
                               media ctrl object.

            _IsQuery        ...True if call is to query sub-system for operation capability

            _IsBlocking     ...True if call is blocking & will not return till complete, or false if
                               non-blocking & returns immediately as "pending".

            MEDIA CONTROL OBJECT SETTINGS & PARAMETERS

            < _Setting >    < _Param >

            BASE OBJECT SETTINGS:
            NoSetting       ...Don't care
            Open            ...Don't care
            Close           ...Don't care
            Enable          ...Don't care
            Disable         ...Don't care
            On              ...Don't care
            Off             ...Don't care
            AttachTo        ...MCO type to which lvalue MCO will be attached
            DetachFrom      ...MCO type to which lvalue MCO will be attached
            DetachAll       ...Don't care
            AddToComposite  ...Ptr to label of MCO to add to lvalue MCO to create composite
            RemoveFromComposite ...Ptr to label of MCO to remove from lvalue composite MCO
            SetCompositeType    ...Value selected from one of < MCO_COMPTYPE >
            GetStatus       ...Adr of Ptr that will point to parameter block appropriate
                               for the lvalue MCO; that is, it will be ptr to one of:
                                        < MCO_AUDIOPARAM >
                                        < MCO_VIDEOPARAM >
                                        < MCO_IMAGEPARAM >
                                        < MCO_DATAPARAM >
            GetCaps         ... < _Param > to whose capability is directed such inquiry

            MEDIA CONTROL OBJECT SETTINGS & PARAMETERS CONTINUED

            MOTION-VIDEO SETTINGS:
            SetColorkey     ... < RGBVALUE > (cast to DWORD argument)
            AssignWindow    ...Ptr to unassigned window's data object
            UnassignWindow  ...Ptr to previously assigned window's data object
            ResizeWindow    ...Ptr to previously assigned window's data object
            SetStretchOn    ...Don't care
            SetStretchOff   ...Don't care
            SetImageType    ...Value selected from one of < IMAGETYPE >
            Freeze          ...Don't care
            Unfreeze        ...Don't care
            SetProportionalOn   ...Don't care
            SetProportionalOff  ...Don't care
            SetVideoFrameSize   ...Value selected from one of < VIDEOSIZE >
```

```
                IMAGING SETTINGS:
4915            AssignWindow          ...Ptr to unassigned window's data object
                UnassignWindow        ...Ptr to previously assigned window's data object
                ResizeWindow          ...Ptr to previously assigned window's data object
                SetStretchOn          ...Don't care
                SetStretchOff         ...Don't care
4920            SetImageType          ...Value selected from one of < IMAGETYPE >
                SetImageMetric        ...Value selected from one of < IMAGEMETRIC >
                SetPixelWidth         ...Integer pixel count
                SetPixelHeight        ...Integer pixel count
                SetPixelDepth         ...Integer pixel count
4925            SetPhysicalWidth      ...Integer units according to current metric
                SetPhysicalHeight     ...Integer units according to current metric
                SetHorzPixelOrigin    ...Integer pixel count (offset from left)
                SetVertPixelOrigin    ...Integer pixel count (offset from top)
                SetHorzPhysicalOrigin ...Integer units according to current metric (offset from left)
4930            SetVertPhysicalOrigin ...Integer units according to current metric (offset from top)
                SetHorzPixelDensity   ...Integer pixel count according to units per current metric
                SetVertPixelDensity   ...Integer pixel count according to units per current metric
                SetImageCombineType   ...Value selected from one of < IMAGECOMBINETYPE >

4935            AUDIO SETTINGS:
                SetAudioQuality       ...Value selected from one of < AUDIOQUALITY >
                SynchToVideo          ...Ptr to video obj label to which lvalue obj will synch
                UnsynchFromVideo      ...Ptr to video obj label from which lvalue obj will be unsynch
                EchoCancelOn          ...Don't care
4940            EchoCancelOff         ...Don't care
                SetDTMFDuration       ...Integer number of milliseconds for duration
                LocalDTMFPulse        ...Integer represenring keypad button
                RemoteDTMFPulse       ...Integer represenring keypad button

4945            DATA SETTINGS:
                SetDataRate           ...Value selected from one of < DATARATE >
                SetSyncXferMode       ...Don't care
                SetAsyncXferMode      ...Don't care
                SetRestrictedMode     ...Don't care
4950            SetUnrestrictedMode   ...Don't care

RETURN: Failure
        Success
        TimedOut
4955    Redundant
        RequestDenied
        NotSupported
        ProcessTerminated
        Capable
4960    Incapable
        MustBeOpened
        Disabled
        InUse
        MemoryAllocError
4965    ResourceAllocError
        InternalError
        TimerFailure
        UndefinedResult
        InvalidDeviceReturn
4970    InvalidOperation
        InvalidOperationNow
        InvalidObject
        InvalidSetting
        InvalidParam
4975    NotEnoughBandwidth
*/
```

```
RESULTCODE SetDefaultMco( MCO_TYPE  _McoType,
                          const char*  _pMcoLabel );
/*
    USAGE:  Set the VCO's default media ctrl object for the specified object type.

    PARAM:  _McoType            ...Type of media ctrl object to which default media ctrl object will be set.

            _pMcoLabel          ...Ptr to label for media ctrl object to set as default.

    RETURN: Failure
            Success
            Redundant
            NotSupported
            MustBeOpened
            Disabled
            InUse
            InternalError
            InvalidDataType
            InvalidObject
            InvalidSetting
*/

RESULTCODE SetDefaultMco( MCO_TYPE  _McoType,
                          HMCO      _hMco );
/*
    USAGE:  Set the VCO's default media ctrl object for the specified object type.

    PARAM:  _McoType            ...Type of media ctrl object to which default media ctrl object will be set.

            _hMco               ...Handle of media ctrl object to set as default.

    RETURN: Failure
            Success
            Redundant
            NotSupported
            MustBeOpened
            Disabled
            InUse
            InternalError
            InvalidDataType
            InvalidObject
            InvalidSetting
*/
```

```
/*-------------------------------
 PROTOCOL MANAGEMENT & CONTROL
 -------------------------------*/

RESULTCODE SetConfProfile( CONFPROFILE _Profile );
/*
    USAGE:  Specify a profile for the conference, or call in progress, or next call  (set current profile); that
            is, select an algorithm that selects device modes most appropriate to the specific call type.

    PARAM:  _Profile                ...Type of conference profile desired.

    RETURN: Failure
            Success
            NotSupported
            Disabled
*/

RESULTCODE SetModes( BASCODE*  _pModeList,
                     int       _nModes = 1 );
/*
    USAGE:  Attempt to set H.221 device modes for call in progress.

    PARAM:  _pModeList          ...Ptr to list (array) of H.221 mode constants to set.

            _nModes             ...Number of modes in list.

    RETURN: Failure
            Success
            MustBeOpened
            Disabled
            InternalError
            InvalidDataType
            InvalidMode
            InvalidParam
            CallMustBeConnected
*/

RESULTCODE SendCaps();
/*
    USAGE:  Transmit local capabilities to remote station. Must be connected or in the process of setting
            up a call.

    PARAM:  none

    RETURN: Failure
            Success
            MustBeOpened
            Disabled
            InternalError
            LineIsDown
            LineNotConnected
*/
```

```
RESULTCODE VerifyBandwidth( BASCODE     _AudioMode,
                            BASCODE     _DataMode );
/*
    USAGE:  Determine if call has sufficient bandwidth to support the specified combination of audio and
            data modes.

    PARAM:  _AudioMode          ...Requested H.221 audio mode.

            _DataMode           ...Requested H.221 data mode.

    RETURN: Failure
            Success
            NotSupported
            MustBeOpened
            Disabled
            InternalError
            TimerFailure
            UndefinedResult
            InvalidMode
            CallMustBeConnected
*/

RESULTCODE SetDeviceTimeout( DWORD _Msec );
/*
    USAGE:  Set default connection timeout value for encapsulated network interface in terms of how long
            it will wait for a response from the network.

    PARAM:  _Msec               ...Timeout value in milliseconds.

    RETURN: Failure
            Success
            MustBeOpened
            Disabled
            InternalError
            TimerFailure
            InvalidParam
*/

RESULTCODE SetConnectTimeout( DWORD _Msec );
/*
    USAGE:  Set default connection timeout value for call controller in terms of how long it will wait for the
            entire call connection sequence to complete.

    PARAM:  _Msec               ...Timeout value in milliseconds.

    RETURN: Failure
            Success
            MustBeOpened
            Disabled
            InvalidParam
*/
```

```
/*-----------------
DATA EXCHANGE CONTROL
-----------------*/

RESULTCODE TransferBuffer( BYTE*      _pBuf,
                           int        _nBytes = 1,
                           const char* _pMcoLabel = 0,
                           BOOL       _IsQuery = 0,
                           BOOL       _IsBlocking = 1 );
/*
    USAGE:  Transfer buffer to or from remote station, depending on the signal type for the object.

    PARAM:  _pBuf          ...Ptr to data buffer to transfer.

            _nBytes        ...Number of bytes to transfer.

            _pMcoLabel     ...Ptr to label for media ctrl data object, or null to indicate default.

            _IsQuery       ...True if call is to query sub-system for operation capability.

            _IsBlocking    ...True if call is blocking & will not return until complete, or false if
                              non-blocking & returns immediately as "pending".

    RETURN: Failure
            Success
            Pending
            TimedOut
            RequestDenied
            NotSupported
            MustBeOpened
            Disabled
            InUse
            MemoryAllocError
            ResourceAllocError
            InternalError
            InvalidDataType
            InvalidObject
            InvalidParam
            CallMustBeConnected
*/
```

```
RESULTCODE TransferObject( MCO_XFEROBJ  _XferObj,
                           XferObject*  _pXferObj,
                           const char*  _pMCOLabel = 0,
                           BOOL         _IsQuery = 0,
                           BOOL         _IsBlocking = 1 );
/*
    USAGE:  Transfer data object to or from remote station, depending on the signal type for the object.

    PARAM:  _XferObj          ...Type of object to transfer.

            _pXferObject      ...Ptr to the object's descriptor object containing specific information.

            _pMcoLabel        ...Ptr to label for media ctrl data object, or null to indicate default.

            _IsQuery          ...True if call is to query sub-system for operation capability.

            _IsBlocking       ...True if call is blocking & will not return until complete, or false if
                                 non-blocking & returns immediately as "pending".

    RETURN: Failure
            Success
            Pending
            TimedOut
            RequestDenied
            NotImplemented
            NotSupported
            MustBeOpened
            Disabled
            InUse
            MemoryAllocError
            ResourceAllocError
            InternalError
            InvalidDataType
            InvalidObject
            InvalidParam
            CallMustBeConnected
*/
```

```
/*------------------
 TERMINAL SERVICE CONTROL
 ------------------*/

RESULTCODE ToTerminal( char* _pFmt, ...);
/*
    USAGE:  Write data to the VCO terminal OUTPUT port optionally using format string and variable
            number of arguments.

    PARAM:  _pFmt                   ...Ptr to format string.

            ...                     ...Variable number of text and data args.

    RETURN: Failure
            Success
            RequestDenied
            Disabled
            InUse
            InvalidDataType
            InvalidParam
*/

RESULTCODE vToTerminal( char*   _pFmt,
                        void*   _pArgList );
/*
    USAGE:  Write data to the VCO terminal OUTPUT port optionally using format string and list of ptrs
            to arguments.

    PARAM:  _pFmt                   ...Ptr to format string.

            _pArgList               ...Ptr to list of ptrs to arg strings.

    RETURN: Failure
            Success
            RequestDenied
            Disabled
            InUse
            InvalidDataType
            InvalidParam
*/
```

```
RESULTCODE FromTerminal( char* _pFmt, ...);
/*
    USAGE:  Write data to the VCO terminal INPUT port (VCO command interpreter) optionally using
            format string and list of ptrs to arguments.

    PARAM:  _pFmt                   ...Ptr to format string.

            ...                     ...Variable number of text and data args.

    RETURN: Failure
            Success
            RequestDenied
            Disabled
            InUse
            InvalidDataType
            InvalidParam
*/

RESULTCODE vFromTerminal( char*   _pFmt,
                          void*   _pArgList );
/*
    USAGE:  Write data to the VCO terminal INPUT port (VCO command interpreter) optionally using
            format string and list of ptrs to arguments.

    PARAM:  _pFmt                   ...Ptr to format string.

            _pArgList               ...Ptr to list of ptrs to arg strings.

    RETURN: Failure
            Success
            RequestDenied
            Disabled
            InUse
            InvalidDataType
            InvalidParam
*/
```

```
RESULTCODE AttachTermToNotifier( HNOTIFIER _hNotifier );
/*
    USAGE:  Attach signal as VCO terminal output device in order to direct terminal OUTPUT port text
            stream to the associated Nonfier Receiver Object (NRO).

    PARAM:  _hNoufier               ..Handle to signal to attach

    RETURN: Failure
            Success
            Redundant
            RequestDenied
            Disabled
            InUse
            InvalidNotifier
*/

RESULTCODE AttachTermToFile( char*      _pFilespec,
                             BOOL       _IsAppend = 0 );
/*
    USAGE:  Attach file as VCO terminal output device in order to direct terminal output port text stream
            to a specific file or file system device.

    PARAM:  _pFilespec              ...Ptr to file specification of file to attach.

            _IsAppend               ...True if new text to be appended to current stream position; false if
                                    stream position to be reset at time of attachement.

    RETURN: Failure
            Success
            Redundant
            RequestDenied
            Disabled
            InUse
            InvalidDataType
*/

RESULTCODE AttachTermToStream( stream*   _pStream,
                               BOOL      _IsAppend = 0 );
/*
    USAGE:  Attach system data stream as VCO terminal output device in order to direct terminal output
            port text stream to specific data stream entity.

    PARAM:  _pStream                ...Ptr to stream to attach.

            _IsAppend               ...True if new text to be appended to current stream position: false if
                                    stream position to be reset at time of attachment.

    RETURN: Failure
            Success
            Redundant
            RequestDenied
            Disabled
            InvalidDataType
*/
```

```
RESULTCODE AttachTermToMco( HMCO   _hMco,
                            BOOL   _IsAppend = 0 );
/*
    USAGE:  Attach media ctrl object as VCO terminal output device in order to direct terminal output
            port text stream to media ctrl object supporting data transfers.

    PARAM:  _hMco               ...Handle to media ctrl object to attach.

            _IsAppend           ...True if new text to be appended to current stream position; false if
                                   stream position to be reset at time of attachment.

    RETURN: Failure
            Success
            Redundant
            RequestDenied
            Disabled
            InvalidDataType
*/

RESULTCODE DetachTermFrom( TERMODEV _OutputDev );
/*
    USAGE:  Remove previously attached signal, file, or data stream from the terminal output port.

    PARAM:  _OutputDev          ...Terminal output device from which to detach terminal output.

    RETURN: Failure
            Success
            Redundant
            RequestDenied
            Disabled
            InUse
            InvalidDataType
*/
```

```
/*--------------------
SYSTEM INFORMATION CONTROL
--------------------*/

    BOOL IsReady();
    /*
        USAGE:  Returns true if VCO is ready to make initial call to remote station or multipoint control unit.

        PARAM:  none

        RETURN: True
                False
    */

    BOOL IsCallSetup();
    /*
        USAGE:  Returns true while call is setting up, but not connected.

        PARAM:  none

        RETURN: True
                False
    */

    BOOL IsCall();
    /*
        USAGE:  Returns true while call is fully connected to remote station.

        PARAM:  none

        RETURN: True
                False
    */

    BOOL IsMultiCall();
    /*
        USAGE:  Returns true while connected to more than one remote station or multipoint control unit.

        PARAM:  none

        RETURN: True
                False
    */

    BOOL IsRemoteVco();
    /*
        USAGE:  Returns true if remote station is a multipoint control unit.

        PARAM:  none

        RETURN: True
                False
    */

    BOOL IsRemoteAttached();
    /*
        USAGE:  Returns true if remote station VCO command and/or event stream is accessible to local station.

        PARAM:  none

        RETURN: True or False
    */
```

```
BOOL IsMultiInstantiated();
/*
    USAGE:  Returns true if this VCO is running with more than one instance.

    PARAM:  none

    RETURN: True or False
*/

DEVICEPARAM& GetDeviceParam();
/*
    USAGE:  Returns reference to static buffer containing copy of VCO device parameters.

    PARAM:  none

    RETURN: Reference to copy of VCO device parameter block
*/

CONFIGPARAM& GetConfigParam();
/*
    USAGE:  Returns reference to static buffer containing copy of VCO config parameters.

    PARAM:  none

    RETURN: Reference to copy of VCO configuration parameter block
*/

CALLPARAM& GetCallParam();
/*
    USAGE:  Returns reference to static buffer containing copy of VCO call parameters.

    PARAM:  none

    RETURN: Reference to copy of VCO call parameter block
*/

PROTOCOLPARAM& GetProtocolParam();
/*
    USAGE:  Returns reference to static buffer containing copy of VCO protocol parameters.

    PARAM:  none

    RETURN: Reference to VCO protocol parameter block
*/

CONTROLPARAM& GetControlParam();
/*
    USAGE:  Returns reference to static buffer containing copy of VCO control context parameters.

    PARAM:  none

    RETURN: Reference to VCO control context parameter block
*/

MONITORPARAM& GetMonitorParam();
/*
    USAGE:  Returns reference to static buffer containing copy of VCO monitor context parameters.

    PARAM:  none

    RETURN: Reference to VCO monitor context parameter block
*/
```

```
VCOPARAM& GetVcoParam();
/*
    USAGE:  Returns reference to static buffer containing copy of all VCO parameters.

    PARAM:  none

    RETURN: Reference to VCO system information parameter block
*/

MCOPARAM& GetMcoParam( MCO_TYPE _McoType );
/*
    USAGE:  Returns reference to static buffer containing copy of media ctrl parameter block.

    PARAM:  _McoType                ..Type of media ctrl object

    RETURN: Reference to media ctrl object parameter block for specified media ctrl object
*/

MCOPARAM& GetMcoParam( HMCO _hMco );
/*
    USAGE:  Returns reference to static buffer containing copy of media ctrl parameter block.

    PARAM:  _hMco                   ...Handle to media ctrl object

    RETURN: Reference to media ctrl object parameter block for specified media ctrl object
*/

VIDEOSIZE GetVideoSize( MCO_SIGNALTYPE _SigType = SignalDst );
/*
    USAGE:  Return video frame size for default video object of specified signal type.

    PARAM:  _SigType                ...Video signal to examine for size

    RETURN: Enumerated video frame size value
*/

AUDIOQUALITY GetAudioQuality( MCO_SIGNALTYPE _SigType = SignalDst );
/*
    USAGE:  Return quality of default audio object of specified signal type.

    PARAM:  _SigType                ...Audio signal to examine for quality

    RETURN: Enumerated audio quality value
*/

const char* GetH221ModeLabel( BASCODE _Mode );
const char* GetH221CapLabel( BASCODE _Cap );
const char* GetDeviceStateLabel( DEVICESTATE _DeviceState );
const char* GetResultCodeLabel( RESULTCODE _ResultCode );
const char* GetEventLabel( EVENT _Event );
const char* GetLineStateLabel( LINESTATE _LineState );
const char* GetCallStateLabel( CALLSTATE _CallState );
const char* GetMediaControlTokenLabel( int _MediaCtrlToken );
const char* GetMcoLabel( HMCO _hMco );
const char* GetMcoLabel( MCO_TYPE _McoType );
const char* GetMultiCallOpLabel( MULTICALLOP _MultiCallOp );
const char* GetStationLabel( BOOL _IsRemoteStation = 0 );
/*
    USAGE:  Return ptr to text label specified state or constant item.

    PARAM:  The specified state or constant

    RETURN: Ptr to constant string if argument valid, else null
*/
```

```
HMCO GetMcoHandle( const char* _pMcoLabel );
/*
    USAGE:  Return handle to media ctrl object specified by its label.

    PARAM:  _pMcoLabel          ...Ptr to label to media ctrl object label.

    RETURN: Handle to media ctrl object specified by the label, else null if no such media ctrl object found.
*/

HMCO GetMcoHandle( MCO_TYPE _McoType );
/*
    USAGE:  Return handle to media ctrl object specified by its type.

    PARAM:  _McoType            ...Media ctrl object type.

    RETURN: Handle to media ctrl object specified by the type, else null if no such media ctrl object found.
*/

RESULTCODE ListDeviceParam();
RESULTCODE ListConfigParam();
RESULTCODE ListCallParam();
RESULTCODE ListProtocolParam();
RESULTCODE ListModeCapsXRefs();
RESULTCODE ListMCOBindings();
RESULTCODE ListMCOs();
RESULTCODE ListNotifiers();
RESULTCODE ListCommands();
RESULTCODE ListTerminalOutputs();
RESULTCODE ListConferees();
RESULTCODE ListConnectionCaps();
RESULTCODE ListMultiCallStates();
/*
    USAGE:  Output formatted text listing for specified parameter groups

    PARAM:  none

    RETURN: Failure
            Success
*/

RESULTCODE ListMcoCaps( MCO_TYPE        _McoType,
                        MCO_SETTING     _Setting );
/*
    USAGE:  Output formatted text listing of capabilities for specified default media ctrl object.

    PARAM:  _McoType            ...Specific media ctrl object to address

            _Setting            ...Audio-video-data setting constant specifying requested service
                                   desired from object.

    RETURN: Failure
            Success
            InvalidObject
            InvalidSetting
*/

RESULTCODE ListStationCaps( STATION* _pStation = 0 );
/*
    USAGE:      Output formatted text listing of H.221 capabilities for specified station.

    PARAM       _pStation       ...Ptr to station descriptor (0 returns caps for local station).

    RETURN: Failure
            Success
            InvalidStation
*/
```

```
/*-------------------------
 VIRTUAL CONNECTION OBJECT CONTROL
 -------------------------*/

RESULTCODE EnableVco( BOOL _IsEnabled = 1 );
/*
    USAGE:  Set VCO enable flag to true or false to change accessibility of VCO.

    PARAM   _IsEnabled              ...True if VCO to be enabled; false to disable.

    RETURN: Failure
            Success
            Redundant
*/

RESULTCODE SetVcoExceptMode( EXCEPTMODE _Mode = ExceptModeUser );
/*
    USAGE:  Set the current VCO exception handling modality.

    PARAM:  _Mode                   ...Exception handling mode desired.

    RETURN: Failure
            Success
            Redundant
            RequestDenied
            NotSupported
*/

RESULTCODE SetVcoTraceMode( TRACEMODE _Mode = 0 );
/*
    USAGE:  Set the current VCO trace output modality.

    PARAM:  _Mode                   ...Trace output mode desired.

    RETURN: Failure
            Success
            Redundant
            RequestDenied
            NotSupported
*/

RESULTCODE EnableMultiCallOps( BOOL _IsEnabled = 1 );
/*
    USAGE:  Enable or disable the multipoint call control operations. This operation can only enable
            multipoint call control operations if they are supported by the VCO implementation.

    PARAM:  _IsEnabled              ...True if multipoint control operations to be enabled; false to disable.

    RETURN: Failure
            Success
            Redundant
            RequestDenied
            NotSupported
            MustBeOpened
            Disabled
            InternalError
*/
```

```
RESULTCODE EnableDispatcher( BOOL _IsEnabled = 1 );
/*
    USAGE:  Enable or disable the VCO dispatcher.

    PARAM:  _IsEnabled          ...True if VCO dispatcher operating; false if not.

    RETURN: Failure
            Success
            Redundant
            InUse
            TimerFailure
*/

RESULTCODE QueueEvent( EVENT    _Id,
                       DWORD    _Param1,
                       WORD     _Param2,
                       STATION* _pStation = 0 );
/*
    USAGE:  Insert event into the event queue.

    PARAM:  _Id                 ...Event identifier.

            _Param1             ...First event parameter.

            _Param2             ...Second event parameter.

            _pStation           ...Ptr to station descriptor (null indicates local station).

    RETURN: Failure
            Success
            QueueFull
            MemoryAllocError
            InvalidStation
            InvalidDataType
            InvalidOperation
*/

RESULTCODE SetDispatcherRate( int _Msec = DefaultDispatcherRate );
/*
    USAGE:  Set the rate at which events are dispatched from the event queue

    PARAM:  _Msec               ...Dispatch rate in milliseconds.

    RETURN: Failure
            Success
            RequestDenied
            TimerFailure
            InvalidParam
*/
```

```
RESULTCODE UpdateCapsList( BASCODE  _Cap,
                           char*    _pCapLabel,
/*                         BOOL     _IsNewCap = 1 );
    USAGE:  Add or remove capability to/from local capability list.

    PARAM:  _Cap            ..Capability constant.

            _pCapLabel      ...Ptr to label for capability.

            _IsNewCaps      ...True if new caps to add to list: false if cap to remove

    RETURN: Failure
            Success
            Redundant
            RequestDenied
            InvalidDataType
            InvalidCapability
            InvalidParam
*/

RESULTCODE UpdateModeCapsXRef( XREF*    _pXRef,
/*                             BOOL     _IsNewXRef = 1 );
    USAGE:  Add or remove mode-caps cross-reference to/from local list.

    PARAM:  _pXRef          ...Ptr to mode-caps cross-reference record.

            _IsNewXRef      ...True if new record to add; false if record to remove.

    RETURN: Failure
            Success
            Redundant
            RequestDenied
            InvalidDataType
            InvalidCapability
*/

RESULTCODE EmuControl( EMUCONTROLOP _Op,
/*                     STATION*     _pStation = 0 );
    USAGE:  Present emulation control operation to a VCO.

    PARAM:  _Op             ...Emulation control operation.

            _pStation       ...Ptr to station descriptor (null indicates local station).

    RETURN: Failure
            Success
            Redundant
            RequestDenied
            NotSupported
            InternalError
            InvalidOperation
            InvalidOperationNow
            InvalidStation
*/
```

```
RESULTCODE AttachToRemoteVco( BOOL _IsMonitorOnly = 1,
                              BOOL _IsBlocking = 1 );
/*
    USAGE:  Gain access to the command and/or event stream of the remote station. This action is only
            possible if the remote station is running a VCO for its multimedia connection services.

    PARAM:  _IsMonitorOnly      ...True if request is only to monitor the remote VCO's event stream,
                                and not its command stream (for the purpose of mastering it).

            _IsBlocking         ...True if call is blocking & will not return until complete, or false if
                                non-blocking & returns immediately as "pending".
    RETURN: Failure
            Success
            Pending
            TimedOut
            Redundant
            RequestDenied
            NotSupported
            ProcessTerminated
            MustBeOpened
            Disabled
            InUse
            InternalError
            UndefinedResult
            CallMustBeConnected
*/

RESULTCODE DetachFromRemoteVco( BOOL _IsBlocking = 1 );
/*
    USAGE:  Discard access gained to command and event stream of remote station.

    PARAM:  _IsBlocking         ...True if call is blocking & will not return until complete, or false if
                                non-blocking & returns immediately as "pending".
    RETURN: Failure
            Success
            Pending
            TimedOut
            Redundant
            RequestDenied
            NotSupported
            ProcessTerminated
            Capable
            Incapable
            MustBeOpened
            InternalError
            TimerFailure
            UndefinedResult
            CallMustBeConnected
*/

RESULTCODE SetVcoControlMode( DWORD _ModeFlags = Master );
/*
    USAGE:  Set the mode of VCO control so that calls to member functions drive the local or the remote
            VCO, as configured.

    PARAM:  _ModeFlags          ...VCO control mode flags selected from < CONTROLMODE >

    RETURN: Failure
            Success
            Redundant
            RequestDenied
            InternalError
            InvalidOperation
            InvalidOperationNow
            CallMustBeConnected
*/
```

```
RESULTCODE SetVcoMonitorMode( DWORD _ModeFlags = MonModeLocal );
/*
    USAGE:  Set the mode of VCO monitoring so that the primary event stream from the VCO emanates
            from the local, remote, or group of stations.

    PARAM:  _ModeFlags          ...VCO monitor mode flags selected from < MONITORMODE >

    RETURN: Failure
            Success
            Redundant
            RequestDenied
            InternalError
            InvalidOperation
            InvalidOperationNow
            CallMustBeConnected
*/

RESULTCODE SetStationLabel( char*   _pLabel,
                            char*   _pNum );
/*
    USAGE:  Set label for the local station.

    PARAM:  _pLabel             ...Ptr to station label.

    RETURN: Failure
            Success
            InvalidDataType
*/
```

```
5895    /*-------------------------------------------------------------------
            ALL MEMBERS BELOW ARE TO BE ACCESSED ONLY FOR THE IMPLEMENTATION
          OF THE ABOVE-DEFINED PUBLIC MEMBER FUNCTIONS THAT COMPRISE THE SOFTWARE CONTROL
                    INTERFACE COMPONENT OF EACH VIRTUAL CONNECTION OBJECT.
                      THEY ARE NOT AVAILABLE TO CLIENT APPLICATIONS.
5900    -------------------------------------------------------------------*/
        private:

        typedef enum {
          AudioSetup,                       // Invoke OEM audio setup component
5905      VideoSetup,                       // Invoke OEM video setup component
          ImageSetup,                       // Invoke OEM image setup component
          DataSetup,                        // Invoke OEM data setup component
          UpdateCapsList,                   // Add or remove device capability
          AttachToRemoteVco,                // Perform device operations to connect to remote VCO
5910      DetachFromRemoteVco,              // Perform device operations to detach from remote VCO
          VendorSpecificOp,                 // Vendor-specific device control operations


5915      IOControlOpEnd
        } IOCONTROLOP;

            /*-------------------------
             PURE VIRTUAL DEVICE CONTROL MEMBERS
5920         -------------------------*/

            virtual RESULTCODE DevOpen( BOOL _IsBlocking = 1) = 0;
            /*
                USAGE:  Open encapsulated devices, load and initialize OEM device control software and MCI device
5925                    drivers; prepare VCO to place outgoing, or receive incoming call.

                PARAM:  _IsBlocking             ...True if call is blocking & will not return until complete, or false if
                                                non-blocking & returns immediately as "pending".

5930            RETURN: Failure
                        Success
                        Pending
                        TimedOut
                        MemoryAllocError
5935                    ResourceAllocError
                        InternalError
                        TimerFailure
                        InvalidDeviceReturn
            */
5940
            virtual RESULTCODE DevClose( BOOL _IsBlocking = 1) = 0;
            /*
                USAGE:  Close encapsulated devices, unload OEM device control software and drivers; shutdown all
                        systems related to establishing calls.
5945
                PARAM:  _IsBlocking             ...True if call is blocking & will not return until complete, or false if
                                                non-blocking & returns immediately as "pending".

                RETURN: Failure
5950                    Success
                        MustBeOpened
                        MemoryAllocError
                        ResourceAllocError
                        InternalError
5955        */
```

```
virtual RESULTCODE DevConnect( CALLPARAM&   _CallParam,
                               STATION*     _pStation,
                               BOOL         _IsBlocking = 1 ) = 0;
/*
    USAGE:  Connect to remote station or multipoint control unit.

    PARAM:  _CallParam      .. Reference to a VCO call parameter record.

            _pStation       ..Ptr to remote station descriptor to which connect is desired.

            _IsBlocking     ...True if call is blocking & will not return until complete. or false if
                               non-blocking & returns immediately as "pending".

    RETURN: Failure
            Success
            Pending
            TimedOut
            MustBeOpened
            InUse
            MemoryAllocError
            ResourceAllocError
            InternalError
            InvalidStation
            InvalidDataType
            LineConnectFailed
            LineIsBusy
*/

virtual RESULTCODE DevMultiConnect( MULTICALLOP _Op,
                                    CALLPARAM&  _CallParam,
                                    STATION*    _pStation = 0,
                                    BOOL        _IsQuery = 0,
                                    BOOL        _IsBlocking = 1 ) = 0;
/*
    USAGE:  Implement multipoint control operation while connected to multipoint control unit.

    PARAM:  _Op             ...Multipoint control operation.

            _CallParam      ...Reference to a VCO call parameter record.

            _pStation       ...Ptr to station descriptor for selected operation.

            _IsQuery        ...True if call is to query sub-system for operation capability.

            _IsBlocking     ...True if call is blocking & will not return until complete. or false if
                               non-blocking & returns immediately as "pending".

    RETURN: Failure
            Success
            Pending
            TimedOut
            Redundant
            RequestDenied
            NotSupported
            MustBeOpened
            InUse
            MemoryAllocError
            ResourceAllocError
            InternalError
            InvalidStation
            InvalidDataType
            InvalidOperation
            InvalidOperationNow
            CallMustBeConnected
            NoCallForLineAdd
*/
```

```
        virtual RESULTCODE DevDisconnect( int     _nLine = 0,
6025                                      BOOL _IsBlocking = 1 ) = 0;
        /*
            USAGE:  Disconnect one or more lines connected to remote station or multipoint control unit.

            PARAM:  _nLine              ...Number of line to disconnect: null disconnects all lines
6030
                    _IsBlocking         ...True if call is blocking & will not return until complete, or false if
                                           non-blocking & returns immediately as "pending".

            RETURN: Failure
6035                Success
                    Pending
                    TimedOut
                    MustBeOpened
                    InUse
6040                MemoryAllocError
                    ResourceAllocError
                    InternalError
                    InvalidLine
                    CallMustBeConnected
6045    */

        virtual RESULTCODE DevAnswer( CALLPARAM&    _CallParam,
                                      int           _nLine ) = 0;
        /*
6050        USAGE:  Answer incoming call from remote station.

            PARAM:  _CallParam          ...Reference to a VCO call parameter record.

                    _nLine              ...Number of line to disconnect: null disconnects all lines.
6055
            RETURN: Failure
                    Success
                    MustBeOpened
                    InUse
6060                MemoryAllocError
                    ResourceAllocError
                    InternalError
                    InvalidDataType
                    InvalidLine
6065                InvalidOperationNow
                    LineConnectFailed
        */

        virtual RESULTCODE DevAbort() = 0;
6070    /*
            USAGE:  Abort entire connection, or connection in progress, to remote station or multipoint control unit.

            PARAM:  none

6075        RETURN: Failure
                    Success
                    TimedOut
                    MustBeOpened
                    InUse
6080                MemoryAllocError
                    ResourceAllocError
                    InternalError
                    CallMustBeConnected
        */
6085
```

```
virtual RESULTCODE DevGetCallInfo( CALLPARAM& _CallParam ) = 0;
/*
    USAGE:  Get information for call, or for partially connected call. Can be used while connection
            establishing to monitor call progress.

    PARAM:  _CallParam              ...Reference to a VCO call parameter record.

    RETURN: Failure
            Success
            TimedOut
            MustBeOpened
            MustBeClosed
            InUse
            MemoryAllocError
            ResourceAllocError
            InternalError
            InvalidDataType
            LineNotConnected
            LineIsBusy
            DisconnectRequest
*/

virtual RESULTCODE DevMediaControl( MCO_TYPE        _McoType,
                                    MCO_SETTING     _Setting,
                                    DWORD           _Param = 0,
                                    BOOL            _IsQuery = 0,
                                    BOOL            _IsBlocking = 1 ) = 0;
/*
    USAGE:  Implement media ctrl control setting by making calls to device control software
            layers and MCI device drivers.

    PARAM:  (same as MediaControl)

    RETURN: Failure
            Success
            Pending
            TimedOut
            RequestDenied
            MustBeOpened
            InUse
            MemoryAllocError
            ResourceAllocError
            InternalError
            TimerFailure
            UndefinedResult
            InvalidDataType
            InvalidOperation
            InvalidOperationNow
            InvalidObject
            InvalidSetting
            InvalidParam
*/
```

```
                virtual RESULTCODE DevEmuControl( EMUCONTROLOP _Op ) = 0;
6140            /*
                    USAGE:  Implement emulation control operation for this VCO.

                    PARAM:  (same as EmuControl)

6145                RETURN: Failure
                            Success
                            Redundant
                            RequestDenied
                            MustBeOpened
6150                        MemoryAllocError
                            ResourceAllocError
                            InternalError
                            InvalidOperation
                            InvalidOperationNow
6155            */
```

```
virtual RESULTCODE DevXmitData( BYTE*  _pBuf,
                                int    _nBytes = 1,
                                HMCO   _hMco = 0,
                                BOOL   _IsQuery = 0,
                                BOOL   _IsBlocking = 1 ) = 0;
/*
    USAGE.  Transmit data buffer to remote station, for a specific data object.

    PARAM:  _pBuf          ...Ptr to buffer containing data.
            _nBytes        ...Number of bytes to transmit.
            _hMco          ...Handle to data object to use for data transfer; null indicates default.
            _IsQuery       ...True if call is to query sub-system for operation capability.
            _IsBlocking    ...True if call is blocking & will not return until complete, or false if
                              non-blocking & returns immediately as "pending".

    RETURN: Failure
            Success
            TimedOut
            NotSupported
            MustBeOpened
            InUse
            MemoryAllocError
            ResourceAllocError
            InternalError
            InvalidDataType
            InvalidObject
            InvalidParam
            CallMustBeConnected
*/

virtual RESULTCODE DevRcvData( BYTE*  _pBuf,
                               int    _nBytes = 1
                               HMCO   _hMco = 0,
                               BOOL   _IsQuery = 0,
                               BOOL   _IsBlocking = 1 ) = 0;
/*
    USAGE:  Post request to receive data buffer from remote station, for a specific data object.

    PARAM:  _pBuf          ...Ptr to buffer containing data.
            _nBytes        ...Number of bytes to transmit.
            _hMco          ...Handle to data object to use for data transfer; null indicates default.
            _IsQuery       ...True if call is to query sub-system for operation capability.
            _IsBlocking    ...True if call is blocking & will not return until complete, or false if
                              non-blocking & returns immediately as "pending".

    RETURN: Failure
            Success
            TimedOut
            NotSupported
            MustBeOpened
            InUse
            MemoryAllocError
            ResourceAllocError
            InternalError
            InvalidDataType
            InvalidObject
            InvalidParam
            CallMustBeConnected
*/
```

```
6225
        virtual RESULTCODE DevSetModes( BASCODE* _pModeList,
                                        int      _nModes = 1 );
        /*
        USAGE:  Attempt to set H.221 device modes for call in progress.
6230
            PARAM:  _pModeList          ...Ptr to list (array) of H.221 mode constants to set.

                    _nModes             ..Number of modes in list.

6235        RETURN: Failure
                    Success
                    TimedOut
                    RequestDenied
                    MustBeOpened
6240                MemoryAllocError
                    ResourceAllocError
                    InternalError
                    InvalidDataType
                    InvalidMode
6245                InvalidParam
                    CallMustBeConnected
        */

        virtual RESULTCODE DevSendCaps( BASCODE* _pCapList,
6250                                    int      _nCaps = 1 ) = 0;
        /*
            USAGE:  Transmit the specified capability set to the remote station.

            PARAM:  (same as SendCaps)
6255
            RETURN: Failure
                    Success
                    TimedOut
                    RequestDenied
6260                MustBeOpened
                    MemoryAllocError
                    ResourceAllocError
                    InternalError
                    InvalidDataType
6265                InvalidCapability
                    InvalidParam
        */
```

```
6270        virtual const char* DevGetModeLabel( BASCODE _Mode ) = 0;
            /*
                USAGE:  Get text label for specified H.221 mode.

                PARAM:  _Mode                   ...H.221 mode constant.

6275            RETURN: Ptr to constant character string if argument valid, else null
            */

            virtual const char* DevGetCapLabel( BASCODE _Cap ) = 0;
            /*
6280            USAGE:  Get text label for specified H.221 capability.

                PARAM:  _Cap                    ...H.221 capability constant.

6285            RETURN: Ptr to constant string if argument valid, else null
            */

            virtual MCOPARAM& DevGetMco( HMCO _hMco ) = 0;
            /*
6290            USAGE:  Return reference to static buffer containing copy of media ctrl object parameter block

                PARAM:  _hMco                   ...Handle to media ctrl object.

                RETURN: Reference to copy of media ctrl object parameter block for specified media ctrl object
            */
6295
            virtual HMCO DevGetMcoHandle( const char* _pMcoLabel ) = 0;
            /*
                USAGE:  Return handle to media ctrl object specified by its label

6300            PARAM:  _pMcoLabel              ...Ptr to label to media ctrl object label.

                RETURN: Handle to media ctrl object specified by the label, else null if no such media ctrl object found.
            */

6305        virtual HMCO DevGetMcoHandle( MCO_TYPE _McoType ) = 0;
            /*
                USAGE:  Return handle to media ctrl object specified by its type.

                PARAM:  _McoType                ...Media ctrl object type.
6310
                RETURN: Handle to media ctrl object specified by the type, else null if no such media ctrl object found.
            */

6315        virtual RESULTCODE DevSetDefaultMco( MCO_TYPE _McoType,
            /*                                       const char*   _pMcoLabel ) = 0;

                USAGE:  Set the VCO's default media ctrl object for the specified object type.

                PARAM:  _McoType                ...Type of media ctrl object to which the default will be set.
6320
                        _pMcoLabel              ...Ptr to label for media ctrl object to set as default.

                RETURN: Failure
                        Success
6325                    Redundant
                        NotSupported
                        MustBeOpened
                        Disabled
                        InUse
6330                    InternalError
                        InvalidDataType
                        InvalidObject
                        InvalidSetting
            */
6335
```

```
virtual RESULTCODE DevSetDefaultMco( MCO_TYPE _McoType,
                                     HMCO     _hMco ) = 0;
/*
    USAGE:  Set the VCO's default media ctrl object for the specified object type.

    PARAM:  _McoType        ...Type of media ctrl object to which default media ctrl object will be set.

            _hMco           ...Handle of media ctrl object to set as default.

    RETURN: Failure
            Success
            Redundant
            NotSupported
            MustBeOpened
            Disabled
            InUse
            InternalError
            InvalidDataType
            InvalidObject
            InvalidSetting
*/

virtual RESULTCODE DevSetTimeout( DWORD _Msec ) = 0;
/*
    USAGE:  Set connect timeout for network interface unit.

    PARAM:  _Msec           ...Timeout value in milliseconds.

    RETURN: Failure
            Success
            TimedOut
            RequestDenied
            MustBeOpened
            MemoryAllocError
            ResourceAllocError
            InternalError
            TimerFailure
            InvalidDataType
            InvalidParam
*/

virtual RESULTCODE DevVerifyBandwidth( BASCODE  _AudioMode,
                                       BASCODE  _DataMode ) = 0;
/*
    USAGE:  Verify that connection has sufficient bandwidth for specified audio-data mode combination,
            with respect to the current video mode (if applicable).

    PARAM:  _AudioMode      ...H.221 audio mode.

            _DataMode       ...H.221 data mode.

    RETURN: TimedOut
            Capable
            Incapable
            MustBeOpened
            InUse
            MemoryAllocError
            ResourceAllocError
            InternalError
            InvalidMode
            CallMustBeConnected
*/
```

```
virtual RESULTCODE DevIOControl( IOCONTROLOP _Op,
                                 DWORD       _Param1 = 0,
                                 DWORD       _Param2 = 0,
                                 BOOL        _IsQuery = 0,
                                 BOOL        _IsBlocking = 1 ) = 0;
/*
    USAGE:  Implement input/output device control operation by making calls to device control software
            layers and specific OEM-provided system components. This member function may be utilized
            by developers to enable customized support for specialized, implementation-dependent
            features that are not well-represented in the standard VCO device control member function
            complement. This member function is provided as a mechanism to build structured,
            easily-documented extensions to the standard pure virtual VCO device control members
            used to implement the public VDI members.

    PARAM:  _Op          ...Input/output device control operation requested.

            _Param1      ...If required, provides parameter necessary to fully specify request.

            _Param2      ...If required, provides parameter necessary to fully specify request

            _IsQuery     ...True if call is to query sub-system for operation capability.

            _IsBlocking  ...True if call is blocking & will not return until complete, or false if
                            non-blocking & returns immediately as "pending".

    RETURN: Failure
            Success
            Pending
            TimedOut
            RequestDenied
            MustBeOpened
            InUse
            MemoryAllocError
            ResourceAllocError
            InternalError
            TimerFailure
            UndefinedResult
            InvalidDataType
            InvalidOperation
            InvalidOperationNow
            InvalidParam
            ...Others according to implementation requirements
*/
```

BIT-RATE ALLOCATION SIGNAL HEADER FILE

```
/*-----------------------------------------------------------------
                        SAMPLE HEADER FILE
                              for
                  H.221 BIT-RATE ALLOCATION SIGNALS
                             used to indicate
                  DEVICE MODES AND CAPABILITIES

                             ABSTRACT

This source module contains header information that will provide a device-independent epresentation of bit-rate allocation signal
commands (BAS codes) used to indicate device modes and capabilities according to the H.320 (specifically H.221)
Recommendation. These lists are intended for illustrative purposes, and are incomplete. An implementation of a VMCS would need
to define a complete list, and then preserve the exact numerical identity of these constants for all implementations intended for
interoperability within the same operating environment. virtualization routines in the VL (of VCO implementations) must translate
these device-independent versions of H.221 modes and capabilities into the actual BAS codes formats specified by the
recommendation for line transmission.

(SOURCE FILE:    BASCODES.H)

                          PROGRAMMING NOTES

This module contains only C++ source code and structured comments using the " // " notation to denote comments (in addition to
the standard C comment notation using "/**/").
-----------------------------------------------------------------*/

/*-------------------------------
  BIT-RATE ALLOCATION SIGNALS TO INDICATE DEVICE
  CAPABILITIES (DEVICE MODE CAPABILITIES)
  ------------------------------- */

// TRANSFER RATE CAPABILITIES
BASCODE CapTransferRate64          = 0x80000001;
BASCODE CapTransferRate2x64        = 0x80000002;
BASCODE CapTransferRate3x64        = 0x80000003;
BASCODE CapTransferRate4x64        = 0x80000004;
BASCODE CapTransferRate5x64        = 0x80000005;
BASCODE CapTransferRate6x64        = 0x80000006;
BASCODE CapTransferRate384         = 0x80000007;
BASCODE CapTransferRate2x384       = 0x80000008;
BASCODE CapTransferRate3x384       = 0x80000009;
BASCODE CapTransferRate4x384       = 0x8000000a;
BASCODE CapTransferRate5x384       = 0x8000000b;
BASCODE CapTransferRate1536        = 0x8000000c;
BASCODE CapTransferRate1920        = 0x8000000d;
BASCODE CapTransferRate128         = 0x8000000e;
BASCODE CapTransferRate192         = 0x8000000f;
BASCODE CapTransferRate256         = 0x80000010;
BASCODE CapTransferRate512         = 0x80000020;
BASCODE CapTransferRate768         = 0x80000030;
BASCODE CapTransferRate1152        = 0x80000040;
BASCODE CapTransferRate1472        = 0x80000050;
BASCODE CapRestrict                = 0x80000060;
BASCODE CapComposite6B             = 0x80000070;

//AUDIO CAPABILITIES
BASCODE CapAudioALaw               = 0x80000080;
BASCODE CapAudioULaw               = 0x80000090;
BASCODE CapAudioG722_64            = 0x800000a0;
BASCODE CapAudioG722_48            = 0x800000b0;
BASCODE CapAudioG728               = 0x800000c0;
BASCODE CapAudioISO                = 0x800000d0;

//VIDEO CAPABILITIES
BASCODE CapVideoQCIF1              = 0x800000e0;
BASCODE CapVideoQCIF2              = 0x800000f0;
```

```
        BASCODE CapVideoQCIF3                   = 0x80000100;
        BASCODE CapVideoQCIF4                   = 0x80000200;
6510    BASCODE CapVideoCIF1                    = 0x80000300;
        BASCODE CapVideoCIF2                    = 0x80000400;
        BASCODE CapVideoCIF3                    = 0x80000500;
        BASCODE CapVideoCIF4                    = 0x80000600;
        BASCODE CapVideo Improved               = 0x80000700;
6515    BASCODE CapVideoISO                     = 0x80000800;
        BASCODE CapVideoComposite               = 0x80000900;

        //IMAGE CAPABILITIES
        BASCODE CapSuntilPictureLowSpeedData    = 0x80000a00;
6520    BASCODE CapSuntilPictureHighSpeedData   = 0x80000b00;
        BASCODE CapSuntilPictureSpatialMode     = 0x80000c00;
        BASCODE CapSuntilPictureProgressiveMode = 0x80000d00;
        BASCODE CapSuntilPictureArithmeticMode  = 0x80000e00;
        BASCODE CapSuntilPictureH261            = 0x80000f00;
6525    BASCODE CapGroup3Fax                    = 0x80001000;
        BASCODE CapGroup4Fax                    = 0x80002000;

        //LOW SPEED DATA CAPABILITIES
        BASCODE CapLowSpeedDataVariable         = 0x80003000;
6530    BASCODE CapLowSpeedData300              = 0x80004000;
        BASCODE CapLowSpeedData1200             = 0x80005000;
        BASCODE CapLowSpeedData4800             = 0x80006000;
        BASCODE CapLowSpeedData6400             = 0x80007000;
        BASCODE CapLowSpeedData8000             = 0x80008000;
6535    BASCODE CapLowSpeedData9600             = 0x80009000;
        BASCODE CapLowSpeedData14400            = 0x8000a000;
        BASCODE CapLowSpeedData16K              = 0x8000b000;
        BASCODE CapLowSpeedData24K              = 0x8000c000;
        BASCODE CapLowSpeedData32K              = 0x8000d000;
6540    BASCODE CapLowSpeedData40K              = 0x8000e000;
        BASCODE CapLowSpeedData48K              = 0x8000f000;
        BASCODE CapLowSpeedData56K              = 0x80010000;
        BASCODE CapLowSpeedData62K              = 0x80020000;
        BASCODE CapLowSpeedData64K              = 0x80030000;
6545

        // HIGH SPEED DATA CAPABILITIES
        BASCODE CapHighSpeedData64K             = 0x80040000;
        BASCODE CapHighSpeedData128K            = 0x80050000;
        BASCODE CapHighSpeedData192K            = 0x80060000;
6550    BASCODE CapHighSpeedData256K            = 0x80070000;
        BASCODE CapHighSpeedData320K            = 0x80080000;
        BASCODE CapHighSpeedData384K            = 0x80090000;
        BASCODE CapHighSpeedData512K            = 0x800a0000;
        BASCODE CapHighSpeedData768K            = 0x800b0000;
6555    BASCODE CapHighSpeedData1152K           = 0x800c0000;
        BASCODE CapHighSpeedData1536K           = 0x800d0000;
        BASCODE CapHighSpeedData                = 0x800e0000;

        //APPLICATION CAPABILITIES
6560    BASCODE CapEncryption                   = 0x800f0000;
        BASCODE CapGraphicsCursor               = 0x80100000;
        BASCODE CapV120LowSpeedData             = 0x80200000;
        BASCODE CapV120HighSpeedData            = 0x80300000;

6565    //MULTIPOINT CONTROL CAPABILITIES (VCO PROPRIETARY)
        BASCODE CapSetConfFocus                 = 0x80400000;
        BASCODE CapQueryConfFocus               = 0x80400001;
        BASCODE CapSetConfChair                 = 0x80400002;
        BASCODE CapQueryConfChair               = 0x80400003;
6570    BASCODE CapAddStation                   = 0x80400004;
        BASCODE CapRemoveStation                = 0x80400005;
        BASCODE CapBroadcastAudio               = 0x80400006;
        BASCODE CapBroadcastVideo               = 0x80400007;
        BASCODE CapBroadcastData                = 0x80400008;
```

```
6575    BASCODE CapGetNumStations                = 0x80400009;
        BASCODE CapGetStationList                = 0x8040000a;
        BASCODE CapGetStationCaps                = 0x8040000b;
        BASCODE CapGetStationAudio               = 0x8040000c;
        BASCODE CapGetStationVideo               = 0x8040000d;
6580    BASCODE CapGetStationData                = 0x8040000e;
        BASCODE CapGetStationIdentity            = 0x8040000f;

        /* ------------------------------------
          BIT-RATE ALLOCATION SIGNALS TO SET DEVICE MODES
6585      (DEVICE MODE COMMANDS)
          ------------------------------------ */

        // TRANSFER RATE MODES
        BASCODE ModeTransferRate64               = 0x10000001;
6590    BASCODE ModeTransferRate2x64             = 0x10000002;
        BASCODE ModeTransferRate3x64             = 0x10000003;
        BASCODE ModeTransferRate4x64             = 0x10000004;
        BASCODE ModeTransferRate5x64             = 0x10000005;
        BASCODE ModeTransferRate6x64             = 0x10000006;
6595    BASCODE ModeTransferRate384              = 0x10000007;
        BASCODE ModeTransferRate2x384            = 0x10000008;
        BASCODE ModeTransferRate3x384            = 0x10000009;
        BASCODE ModeTransferRate4x38             = 0x1000000a;
        BASCODE ModeTransferRate5x384            = 0x1000000b;
6600    BASCODE ModeTransferRate1536             = 0x1000000c;
        BASCODE ModeTransferRate1920             = 0x1000000d;
        BASCODE ModeTransferRate128              = 0x1000000e;
        BASCODE ModeTransferRate192              = 0x1000000f;
        BASCODE ModeTransferRate256              = 0x10000010;
6605    BASCODE ModeTransferRate512              = 0x10000020;
        BASCODE ModeTransferRate768              = 0x10000030;
        BASCODE ModeTransferRate1152             = 0x10000040;
        BASCODE ModeTransferRate1472             = 0x10000050;

6610    // AUDIO MODES
        BASCODE ModeAudioOff_1                   = 0x00000001;
        BASCODE ModeAudioOff_2                   = 0x00000002;
        BASCODE ModeAudioALaw_1                  = 0x00000003;
        BASCODE ModeAudioALaw_2                  = 0x00000004;
6615    BASCODE ModeAudioALaw_3                  = 0x00000005;
        BASCODE ModeAudioULaw_1                  = 0x00000006;
        BASCODE ModeAudioULaw_2                  = 0x00000007;
        BASCODE ModeAudioULaw_3                  = 0x00000008;
        BASCODE ModeAudioG722_1                  = 0x00000009;
6620    BASCODE ModeAudioG722_2                  = 0x0000000a;
        BASCODE ModeAudioG722_3                  = 0x0000000b;
        BASCODE ModeAudio40K                     = 0x0000000c;
        BASCODE ModeAudio32K                     = 0x0000000d;
        BASCODE ModeAudio24K                     = 0x0000000e;
6625    BASCODE ModeAudio8K                      = 0x0000000f;
        BASCODE ModeAudio64K                     = 0x00000010;
        BASCODE ModeAudio128K                    = 0x00000020;
        BASCODE ModeAudio192K                    = 0x00000030;
        BASCODE ModeAudio256K                    = 0x00000040;
6630    BASCODE ModeAudio384K                    = 0x00000050;

        // VIDEO MODES
        BASCODE ModeVideoOff                     = 0x20000001;
        BASCODE ModeVideoH261                    = 0x20000002;
6635    BASCODE ModeVideoImproved                = 0x20000003;
        BASCODE ModeVideoISO                     = 0x20000004;
        BASCODE ModeVideoComposite               = 0x20000005;
        BASCODE ModeVideoCIF                     = 0x20000006;
        BASCODE ModeVideoQCIF                    = 0x20000007;
6640    BASCODE ModeVideo4CIF                    = 0x20000008;
        BASCODE ModeVideoCIF240                  = 0x20000009;
```

```
BASCODE ModeVideoFreeze                          = 0x2000000a;
BASCODE ModeVideoUnfreeze                        = 0x2000000b;
BASCODE ModeVideoFastUpdate                      = 0x2000000c;
BASCODE ModeVideoDecOn                           = 0x2000000d;
BASCODE ModeVideoDecOff                          = 0x2000000e;
BASCODE ModeVideoSplitOn                         = 0x2000000f;
BASCODE ModeVideoSplitOff                        = 0x20000010;

//IMAGE MODES
BASCODE ModeISOSuntilPictureLowSpeedData         = 0x20000020;
BASCODE ModeISOSuntilPictureHighSpeedData        = 0x20000030;
BASCODE ModeLowSpeedDataFax                      = 0x20000040;
BASCODE ModeHighSpeedDataFax                     = 0x20000050;
BASCODE ModeJPEGLowSpeedData                     = 0x20000060;
BASCODE ModeJPEGHighSpeedData                    = 0x20000070;

//LOW SPEED DATA MODES
BASCODE ModeLowSpeedDataOff                      = 0x30000001;
BASCODE ModeLowSpeedData300                      = 0x30000002;
BASCODE ModeLowSpeedData1200                     = 0x30000003;
BASCODE ModeLowSpeedData4800                     = 0x30000004;
BASCODE ModeLowSpeedData6400                     = 0x30000005;
BASCODE ModeLowSpeedData8000                     = 0x30000006;
BASCODE ModeLowSpeedData9600                     = 0x30000007;
BASCODE ModeLowSpeedData14400                    = 0x30000008;
BASCODE ModeLowSpeedData16K                      = 0x30000009;
BASCODE ModeLowSpeedData24K                      = 0x3000000a;
BASCODE ModeLowSpeedData32K                      = 0x3000000b;
BASCODE ModeLowSpeedData40K                      = 0x3000000c;
BASCODE ModeLowSpeedData48K                      = 0x3000000d;
BASCODE ModeLowSpeedData56K                      = 0x3000000e;
BASCODE ModeLowSpeedData62K                      = 0x3000000f;
BASCODE ModeLowSpeedData64K                      = 0x30000010;
BASCODE ModeLowSpeedDataVariable                 = 0x30000020;

// HIGH SPEED DATA MODES
BASCODE ModeHighSpeedDataOff                     = 0x30000030;
BASCODE ModeHighSpeedData64K                     = 0x30000040;
BASCODE ModeHighSpeedData128K                    = 0x30000050;
BASCODE ModeHighSpeedData192K                    = 0x30000060;
BASCODE ModeHighSpeedData256K                    = 0x30000070;
BASCODE ModeHighSpeedData320K                    = 0x30000080;
BASCODE ModeHighSpeedData384K                    = 0x30000090;
BASCODE ModeHighSpeedData512K                    = 0x300000a0;
BASCODE ModeHighSpeedData768K                    = 0x300000b0;
BASCODE ModeHighSpeedData1152K                   = 0x300000c0;
BASCODE ModeHighSpeedData1536K                   = 0x300000d0;
BASCODE ModeHighSpeedDataVariable                = 0x300000e0;

// APPLICATION MODES
BASCODE ModeNeutral                              = 0x20000060;
BASCODE ModeEncryptionOn                         = 0x20000070;
BASCODE ModeEncryptionOff                        = 0x20000080;
BASCODE ModeAudioLoopback                        = 0x20000090;
BASCODE ModeVideoLoopback                        = 0x200000a0;
BASCODE ModeRestrictOn                           = 0x200000b0;
BASCODE ModeRestrictOff                          = 0x200000c0;
BASCODE ModeDigitalLoopback                      = 0x200000d0;
BASCODE ModeLoopbackOff                          = 0x200000e0;
BASCODE ModeComposite6BOn                        = 0x200000f0;
BASCODE ModeComposite6BOff                       = 0x20000100;
BASCODE ModeCursorOnLowSpeedData                 = 0x20000200;
BASCODE ModeFaxOnLowSpeedData                    = 0x20000300;
BASCODE ModeFaxOnHighSpeedData                   = 0x20000400;
BASCODE ModeVI20LowSpeedData                     = 0x20000500;
BASCODE ModeVI20HighSpeedData                    = 0x20000600;
```

PHYSICAL DEVICE INTERFACE HEADER FILE

```
/*-----------------------------------------------------------------------
                    PHYSICAL DEVICE INTERFACE HEADER FILE
                                  for
                VIRTUALIZED MULTIMEDIA CONNECTION SYSTEMS

                                ABSTRACT
This source module contains header information used primarily by the server components in any Virtualized Multimedia Connection
System (VMCS) implementation. If the special keyword symbol "_VCO_BUILD" is defined prior to inclusion of this file, it
indicates to the compiler that a VCO is being built, and the class "VL" must be defined in full. If this symbol is not defined, it
indicates that a VCO client application is being built, and only the header files needed to access members of class VDI, and its pure
virtual device control override member functions in class "PDI", need be considered during the software build process. In this way,
both the server (VCO) and client components of the VMCS derive symbolic definitions from the same source code base, but no
vendor-specific (device-dependent) code is at any time visible to the device-independent client applications.

(SOURCE FILE:    PDI.H)

                            PROGRAMMING NOTES
1. This module contains only C++ source code and structured comments using the " // " notation to denote comments (in addition
to the standard C comment notation using " /* */ ").

2. Symbols defined in the VDI Software Control Interface are shown in boldface type below.
-----------------------------------------------------------------------*/
#include   < OS.H >                    // Include operating system and user interface API
#include   < BASCODES.H >              // Include bit-rate allocation signal indications
#include   < MCI.H >                   // Include Media Control Interface device control constants and structs
#include   <VDI.H>                     // Include definition for the VDI and all less derived classes

/*-------------------
 DECLARATION FOR CLASS VL
 -------------------*/

class VL: public VDI {
    protected:
        VL(const char* _IniFile);

        virtual ~VL();

        virtual const char* GetClassName() { return "VL"; };

    #ifdef _VCO_BUILD

      /*  Vendor-specific special purpose members needed to implement more-derived PDI
          members are defined here. These functions must provide all services necessary to
          format data and control devices in a way consistent with those necessary to best implement
          the overrides to the pure virtual device control members in the VDI.
        */
    #endif
};

/*-------------------
 DECLARATION FOR CLASS PDI
 -------------------*/

class PDI: public VL {
    private:

        // PDI DATA STRUCTURES
        char*       pLabel;          // Ptr to VCO label string
        char*       pVersion;        // Ptr to VCO version string
        DEVCAPS     Local;           // Local device capabilities listing
        int         nModes;          // Number of entries in "Modes to Caps" xref list
        int         nCaps;           // Number of entries in "Caps to Modes" xref list
        XREF        Caps[MaxCaps];   // "Caps to Modes" xref list
        XREF        Modes[MaxModes]; // "Modes to Caps" xref list
```

```
6775        const int       Devices;        // Number of encapsulated devices
            const DEVICE    Dev[MaxDevices]; // Encapsulated device chain
            int             nMco;           // Number of media ctrl objects currently available
            int             nAudioObj;      // Number of audio objects currently available
            int             nVideoObj;      // Number of mooon-video objects
6780        int             nImageObj;      // Number of objects
            int             nDataObj;       // Number of data objects
            const char*     pMediaLabel[];  // Ptr to array of ptrs to media ctrl object labels
            MCO_BINDING     pMediaBinding;  // Ptr to linked list of current media ctrl object bindings
            HMCO*           phMco;          // Ptr to linked list of all available media ctrl objects
6785        HMCO            hMco[MaxMcoType]; // Default media ctrl object handles

            PDI(const char* _IniFile);

            virtual ~PDI();

6790        virtual const char* GetClassName() { return "PDI"; };

            // PURE VIRTUAL OVERRIDES FOR VDI DEVICE CONTROL MEMBERS
            RESULTCODE DevOpen( BOOL);
6795        RESULTCODE DevClose( BOOL);

            RESULTCODE DevConnect( CALLPARAM&,STATION*,BOOL );
            RESULTCODE DevMultiConnect( MULTICALLOP,CALLPARAM&,STATION,BOOL,BOOL );
            RESULTCODE DevDisconnect( int,BOOL );
6800        RESULTCODE DevAnswer( CALLPARAM&,int );
            RESULTCODE DevAbort();
            RESULTCODE DevGetCallInfo( CALLPARAM& );

            RESULTCODE DevMediaControl( MCO_TYPE,MCO_SETTING,DWORD,BOOL,BOOL );
6805
            RESULTCODE DevEmuControl( EMUCONTROLOP );

            RESULTCODE DevXmtData( BYTE*, int, HMCO,BOOL,BOOL );
            RESULTCODE DevRcvData( BYTE*,int,HMCO,BOOL,BOOL );
6810
            RESULTCODE DevSetModes( BASCODE*,int );
            RESULTCODE DevSendCaps( BASCODE*, int );

            const char* DevGetModeLabel( BASCODE );
6815        const char* DevGetCapLabel( BASCODE );

            MCOPARAM& DevGetMco( HMCO );
            HMCO DevGetMcoHandle( const char* );
            HMCO DevGetMcoHandle( MCO_TYPE );
6820
            RESULTCODE DevSetDefaultMco( MCO_TYPE,const char* );
            RESULTCODE DevSetDefaultMco( MCO_TYPE,HMCO );

            RESULTCODE DevSetConfig( CONFIGPARAM& );
6825        RESULTCODE DevGetConfig( CONFIGPARAM& );

            RESULTCODE DevSetTimeout( DWORD );

            RESULTCODE DevVerifyBandwidth( BASCODE,BASCODE );
6830
            RESULTCODE DevIOControl( IOCONTROLOP,DWORD,DWORD,BOOL,BOOL );

            // CALLBACK MEMBERS ACCESSED BY PROTCOL STACK AND MAC LAYER

6835        void far pascal NetworkEvent( DWORD EventCode, DWORD _Param );

            void far pascal DeviceEvent( MCIHEADER* _pMciHeader );
        };
```

GENERAL VMCS HEADER FILE

```
/*-----------------------------------------------------------------
                        GENERAL HEADER FILE
                              for
                VIRTUALIZED MULTIMEDIA CONNECTION SYSTEMS

                             ABSTRACT
This source module contains header information used by both client and server components in any Virtualized Multimedia
Connection System (VMCS) implementation. In this way, both the server (VCO) and client components of the VMCS derive
symbolic definitions from the same source code base. This class serves as a capstone to the VCO class structure: so as to present
every VCO client with exactly the same member functions accessible from the same class type. The addition of implementation-
specific members to this class can proceed without effecting VCO interoperability or standard VMCS documentation.

(SOURCE FILE:    VCO.H)

                          PROGRAMMING NOTES
This module contains only C++ source code and structured comments using the " // " notation to denote comments (in addition to
the standard C comment notation using " /* */ ").
-----------------------------------------------------------------*/

#include   < PDI.H >                    // Include definition for the PDI and all less derived classes

/*--------------------
 DECLARATION FOR CLASS VCO
 --------------------*/

class VCO: public PDI {

    public:

        VCO(const char* _IniFile);

        virtual ~VCO();

        virtual const char* GetClassName() { return "VCO"; };

    private:
        /*   Implementation-specific members go here, including members to support :
                ...Dynamic link library implementation
                ...Access restriction for VCO Clients
                ...Safeguards against re-entrancy and multi-instantiation
         */
};
```

SAMPLE VCO CLIENT APPLICATION

```
/*-------------------------------------------------------------------
                                SAMPLE
                        VIRTUAL CONNECTION OBJECT
                           CLIENT APPLICATION
                                 for
                   VIRTUALIZED MULTIMEDIA CONNECTION SYSTEMS

                                ABSTRACT

This source module contains code to create a simple VCO client application that establishes a concurrent media ctrl connectivity
session. If the selected VCO is found to be capable of concurrent media ctrl connections, it connects to a default remote station
whose numbers are stored in an initialization file. After successful connection, all incoming signals from the remote station are
looped back to it; incoming audio and video signals from the remote station are monitored locally. For clarity, it is assumed that
both the local and remote stations are capable of these operations, and the remote station is actively transmitting audio, video, and
data signals to the local station. Requisite error checking has omitted for most operations.

                (SOURCE FILE:    VCOCLIENT.CPP)

                            PROGRAMMING NOTES

1. This module contains only C++ source code and structured comments using the ' // ' notation to denote comments (in addition
to the standard C comment notation using ' /* */ ').

2. Details related to operating system API's, and the specifics of the user interface, have been omitted for clarity.

3. Symbols defined in the VDI Software Control Interface are shown in boldface type below.
-------------------------------------------------------------------*/

#include   < VCO.H >                          // Include standard VCO Software Control Interface definition

#define    NONBLOCK   0                       // Used to set calls to "non-blocking" mode (immediate return)

/*  Preprocessor macro to create "thunk" that enables the VCO to call Notifier Receiver Object class members
    transparently, in order to notify them that an event has taken place that has triggered the signal.
*/
#define    NOTIFICATION_RECEIVER_MEMBER( _Class, _Member ) \

    #define    EVENTPROC Notifier##_Class##_Member(   EVENT      _Id,
                                                      DWORD      _Param1,
                                                      DWORD      _Param2,
                                                      STATION*   _pStation,
                                                      HNOTIFIER  _hNotifier ) { \
        _Class* pNRO; \
        return ( pNRO ? pNRO-> _Member( _Id, _Param1, _Param2, _pStation, _hNotifier ) : 0 ); \
    }


/*  Preprocessor macro used to specify the internal "thunk" procedure name to the VCO, such as when creating
    a new signal object.
*/
#define    NOTIFICATION_PROC( _Class, _Member )   Notifier##_Class##_Member
```

```
/*  VCO Conferencing Object Class used by client application to establish automatic media ctrl loopback
    connection to remote station. Logs all relevent connection and media control trace information to a file. Also,
    all trace information emanating from the VCO itself, relating to the operations of the VCO, are displayed (in
    real-time) on the console display.
*/
class ConfObject {

    public:

        BOOL IsActive;                          // True if the current session is active

        // Constructor to establish session with remote station (initiate call, then set loopback)
        ConfObject(VCO& _Vco, char* _pTraceFile);

        // Destructor ends session with remote station (hangup)
        ~ConfObject();

    private:

        VCO&    Vco;                            // Reference to default VCO for session
        char*   pTraceFile;                     // Ptr to filespec for session trace file

        HNOTIFIER hNotifyNotifier;              // Handle for event notification signal
        HNOTIFIER hDisplayNotifier;             // Handle for console message display signal

        // Notifier handling procedure for connection and VCO events transmitted to this client class object
        DWORD NotifyProc(EVENT      _Id,
                         DWORD      _Param1,
                         DWORD      _Param2,
                         STATION*   _pStation,
                         HNOTIFIER  _hNotifier );

        // Notifier handling procedure to display the VCO text stream transmitted to this client class object
        DWORD DisplayProc(EVENT     _Id,
                          DWORD     _Param1,
                          DWORD     _Param2,
                          STATION*  _pStation,
                          HNOTIFIER _hNotifier );
};

/*  Create "thunks" for all the VCO event handling procedures used to direct trigger notifications (and text
    messages) to members in the "ConfObject" notification receiver object (NRO).
*/
NOTIFICATION_RECEIVER_MEMBER( ConfObject, NotifyProc);
NOTIFICATION_RECEIVER_MEMBER( ConfObject, DisplayProc);

ConfObject::ConfObject(VCO& _Vco, char* _pTraceFile) {

    /*  Determine if constructed VCO supports device modes necessary to a conference
        where audio, video, and binary information will be concurrently shared.
    */
    IsActive        = 0;
    Vco             = _Vco;
    pTraceFile      = _pTraceFile;
    hNotifyNotifier = 0;
    hDisplayNotifier= 0;
    BOOL CanDoAudio = 0;
    BOOL CanDoVideo = 0;
    BOOL CanDoData  = 0;
    DWORD EventMask = NewRcvMode | NewXmitMode | NewVcoState | NewCallState;
    DEVCAPS&LocalCaps = Vco.GetDeviceParam().Caps.Local;
```

```
    for ( int i = 0; i < LocalCaps.nCaps; i++ ) {
        if (LocalCaps.Cap[i] == CapVideoQCIF1)          // Capable of video mode ?
            CanDoVideo = 1;
        if (LocalCaps.Cap[i] == CapAudioALaw)           // Capable of audio mode ?
            CanDoAudio = 1;
        if (LocalCaps.Cap[i] == CapHighSpeedData384K)   // Capable of data mode ?
            CanDoData = 1;
    }

    /*  If media ctrl modes supported, open VCO for usage:
        setup notifications and then initialize devices
    */
    if (CanDoAudio && CanDoVideo && CanDoData) {

        Vco.NewNotifier(    hNotifyNotifier,
                            NOTIFICATION_PROC(ConfObject, NotifyProc),
                            this,
                            EventMask );
        Vco.EnableNotifier( hDisplayNotifier );

        Vco.NewNotifier(    hDisplayNotifier,
                            NOTIFICATION_PROC(ConfObject, DisplayProc),
                            this,
                            NewTermOutput );

        // Activate the sending of VCO messages to the display console
        Vco.AttachTermToNotifier( hDisplayNotifier );
        Vco.EnableNotifier( hDisplayNotifier );

        IsActive = 1;                               // Mark this session as currently active

        Vco.Open( NONBLOCK );                       // Initialize and activate encapsulated sub-system
    }

    // Otherwise indicate failure to support operations, and exit.
    else printf( "Selected VCO incapable of concurrent media ctrl session.\n" );
}

ConfObject::~ConfObject() {

    // If currently in call, hang it up and output message to trace file
    if ( Vco.IsCall() ) {
        Vco.ToTerminal( "Hanging up call in progress prior to close.\n" );
        Vco.Hangup();
    }

    Vco.Close();                                    // Shutdown the encapsulated sub-system (wait until complete)

    // Delete the notifications
    Vco.DeleteNotifier( hNotifyNotifier );
    Vco.DeleteNotifier( hDisplayNotifier );

    printf("VCO has been closed.\n");
}

DWORD ConfObject::DisplayProc(  EVENT       _Id,
                                DWORD       _Param1,
                                DWORD       _Param2,
                                STATION*    _pStation,
                                HNOTIFIER   _hNotifier ) {

    printf( "%s\n",(char*)_Param2 );                // Display the text message on the console (std output)
}
```

```
DWORD ConfObject::NotifyProc(   EVENT       _Id,
                                DWORD       _Param1,
                                DWORD       _Param2,
                                STATION*    _pStation,
                                HNOTIFIER   _hNotifier ) {

    // Process all events that trigger notification
    switch ( _Id ) {

    case NewRcvMode:                        // Log new mode set by remote station
        Vco.ToTerminal( "Mode Set by RemoteStation [ %s ]\n",
                        Vco.GetModeLabel((BASCODE)_Param1) );
        break;

    case NewXmtMode:                        // Log new mode set by local station
        Vco.ToTerminal( "Mode Set by Local Station [ %s ]\n",
                        Vco.GetModeLabel((BASCODE)_Param1) );
        break;

    case NewVcoState:                       // Handle new VCO state
        switch ( (int)_Param1 ) {
        case VcoOpen:                       // Call default remote station when opened
            Vco.ToTerminal( "Successful VCO open; Calling default remote station.\n" );
            Vco.Call( 0, 0, NONBLOCK);
            break;

        case VcoClose:                      // Log new VCO close state, then mark session inactive
            Vco.ToTerminal( "VCO has been closed.\n" );
            IsActive = 0;
            break;

        case VcoFailed:                     // Log VCO error state, then mark session inactive
        case VcoDisabled:
            Vco.ToTerminal( "VCO Error Condition [%s]\n",
                            Vco.GetVcoStateLabel((int)_Param1));
            IsActive = 0;
            break;
        default:
            break;
        }

    case NewCallState:                      // Handle new VCO call state
        switch ( (int)_Param1) {
        case CallDisconnected:              // Log disconnect of call; end output to trace file; mark session inactive
            Vco.ToTerminal( "Disconnected from Remote Station.\n" );
            Vco.DetachTermFrom( TermODevFile );
            IsActive = 0;
            break;

        case CallConnecting:                // Begin trace file output; trace start of call connection events
            Vco.AttachTermToFile( pTraceFile) ;
            Vco.ToTerminal( "Connecting To Remote Station.\n" );
            break;

        case CallConnected:                 // Upon connection, trace formatted session information to file
            Vco.ToTerminal( "Connected to Remote Station; Listing connection data.\n" );
            Vco.ListCallParam();
            Vco.ListMCOs();
            Vco.ListConnectionCaps();

            // Loop audio, video, and data input signals back to remote station
            Vco.ToTerminal( "Setting up media ctrl loopback...\n" );
            Vco.MediaControl( AudioIn, AttachTo, AudioOut );
            Vco.MediaControl( VideoIn, AttachTo, VideoOut );
            Vco.MediaControl( DataIn, AttachTo, DataOut );
```

```
                    // Set local audio and video (mic and display) to monitor input signals from the remote station
7130                Vco.MediaControl( AudioIn, AttachTo, AudioDst );
                    Vco.MediaControl( VideoIn, AttachTo, VideoDst );
                    break;
                default:
                    break;
7135            }
                break;
            default:
                break;
            }
7140        return 0;
        }

        /*  VCO Client Application to call remote host. Loops back all audio, video, and data channels when connected;
            writes trace of diagnostic session information to backing store in real-time.
7145    */

        main() {

            // Construct a selected VCO
7150        MyVco Vco( "C:\VCO.INI" );

            // Construct the conferencing object
            MyClientApp ConfObject( MyVco, "C:\VCO.LOG" );

7155        // Block while the connectivity session is active
            while ( MyClientApp.IsActive );
        }
```

What is claimed is:

1. A multimedia connectivity program residing in computer readable memory, said connectivity program when executed on a computer providing to an application program multimedia connectivity services through a real-time multimedia device control subsystem including components selected from among a plurality of multimedia devices and a plurality of real-time multimedia protocol stacks, said program comprising:

a single binary object encapsulating a virtual device interface and a device control interface, said virtual device interface including a plurality of virtual methods that represent logical operations available to the application program for controlling said multimedia device control subsystem, said plurality of virtual functions being completely independent of the components within the device control subsystem, said device control interface mapping said plurality of virtual functions to physical control methods which control the components of the multimedia control subsystem.

2. The multimedia connectivity program of claim 1 wherein said device control interface comprises a plurality of media control objects which represent audiovisual and binary data streams associated with the components of the plurality of devices and/or real-time multimedia protocol stacks.

3. The multimedia connectivity program of claim 1 wherein the virtual device interface is configured to present a logical representation of the multimedia connectivity services provided by the connectivity program.

4. The multimedia connectivity program of claim 1 wherein said device control interface comprises a virtualization layer and a physical device interface layer, said virtualization layer located between said virtual device interface and said physical device interface, said physical device interface directly interfacing to the device control subsystem to provide a physical implementation of services requested by the application through the virtual device interface, said virtualization layer residing between the virtual device interface and the physical device interface layer and configured to translate and map device control mechanisms employed by the underlying multimedia control sub-system to representations required by the virtual methods of the virtual device interface.

5. The multimedia connectivity program of claim 2 wherein the plurality of media control objects provides the multimedia connectivity control program with a pool of media device signal resources.

6. The multimedia connectivity program of claim 5 wherein each of said plurality of media control objects is classified as at least one of type of the group consisting of an audio type, a video type, an image type, and a binary data type.

7. The multimedia connectivity program of claim 6 wherein each of said plurality of media control objects represents a signal from the group consisting of a signal from a remote station, a signal to a remote station, a signal from a local output device, and a signal to a local output device.

8. The multimedia connectivity program of claim 7 wherein operations performed on the plurality of media control objects by the physical device layer under control of the virtual device interface implements a logical software switching mechanism connecting incoming signal paths to outgoing signal paths.

9. The multimedia connectivity program of claim 1 wherein the virtual device interface implements a plurality of public member functions, said virtual functions being a subset of those public member functions and wherein said plurality of public member functions represents all of the public member functions in the single binary object that are accessible by the application program.

10. A computer programmed to provide to an application program multimedia connectivity services through a real-time multimedia device control subsystem, the multimedia device control subsystem including components selected from among a plurality of multimedia devices and a plurality of real-time multimedia protocol stacks, said programmed computer comprising:

a virtual device interface and a device control interface, both of which are encapsulated in a single binary object, said virtual device interface including a plurality of virtual methods that represent logical operations available to the application program for controlling said multimedia device control subsystem, said plurality of virtual functions being completely independent of the components within the device control subsystem, said device control interface mapping said plurality of virtual functions to physical control methods which control the components of the multimedia control subsystem.

11. A computer implemented method of providing multimedia connectivity services through a real-time multimedia device control subsystem, the multimedia device control subsystem including components selected from among a plurality of multimedia devices and a plurality of real-time multimedia protocol stacks, said method comprising:

defining and supporting by computer implemented steps a virtual device interface; and

defining and supporting by computer implemented steps a device control interface, wherein both of said virtual device interface and said device control interface are encapsulated in a single binary object, said virtual device interface including a plurality of virtual methods that represent logical operations available to the application program for controlling said multimedia device control subsystem, said plurality of virtual functions being completely independent of the components within the device control subsystem, said device control interface mapping said plurality of virtual functions to physical control methods which control the components of the multimedia control subsystem.

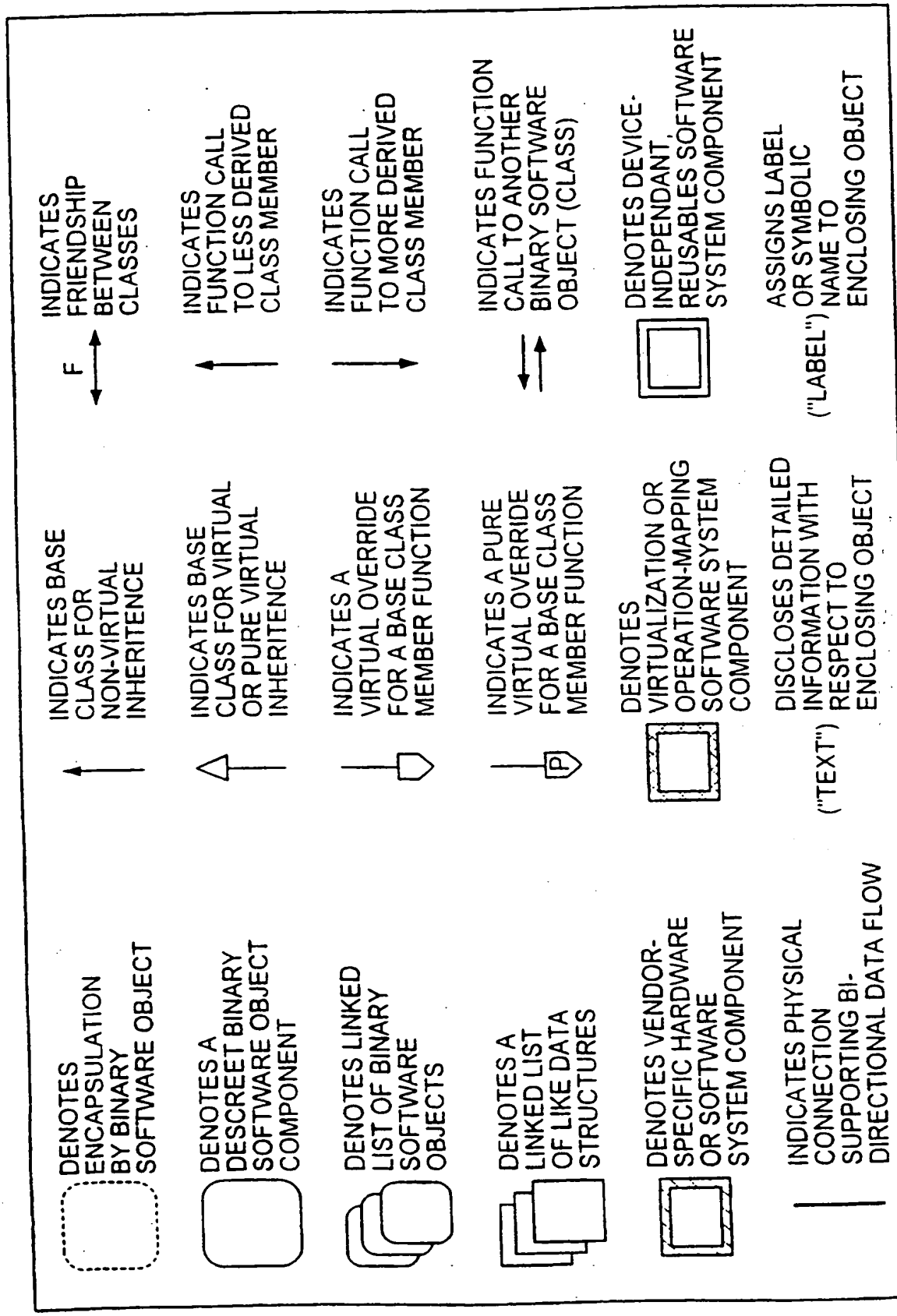

DENOTES ENCAPSULATION BY BINARY SOFTWARE OBJECT
DENOTES A DESCREET BINARY SOFTWARE OBJECT COMPONENT
DENOTES LINKED LIST OF BINARY SOFTWARE OBJECTS
DENOTES A LINKED LIST OF LIKE DATA STRUCTURES
DENOTES VENDOR-SPECIFIC HARDWARE OR SOFTWARE SYSTEM COMPONENT
INDICATES PHYSICAL CONNECTION SUPPORTING BI-DIRECTIONAL DATA FLOW
INDICATES BASE CLASS FOR NON-VIRTUAL INHERITENCE
INDICATES BASE CLASS FOR VIRTUAL OR PURE VIRTUAL INHERITENCE
INDICATES A VIRTUAL OVERRIDE FOR A BASE CLASS MEMBER FUNCTION
INDICATES A PURE VIRTUAL OVERRIDE FOR A BASE CLASS MEMBER FUNCTION
P
DENOTES VIRTUALIZATION OR OPERATION-MAPPING SOFTWARE SYSTEM COMPONENT
("TEXT") DISCLOSES DETAILED INFORMATION WITH RESPECT TO ENCLOSING OBJECT
F
INDICATES FRIENDSHIP BETWEEN CLASSES
INDICATES FUNCTION CALL TO LESS DERIVED CLASS MEMBER
INDICATES FUNCTION CALL TO MORE DERIVED CLASS MEMBER
INDICATES FUNCTION CALL TO ANOTHER BINARY SOFTWARE OBJECT (CLASS)
DENOTES DEVICE-INDEPENDANT, REUSABLES SOFTWARE SYSTEM COMPONENT
("LABEL") ASSIGNS LABEL OR SYMBOLIC NAME TO ENCLOSING OBJECT

FIG. 1

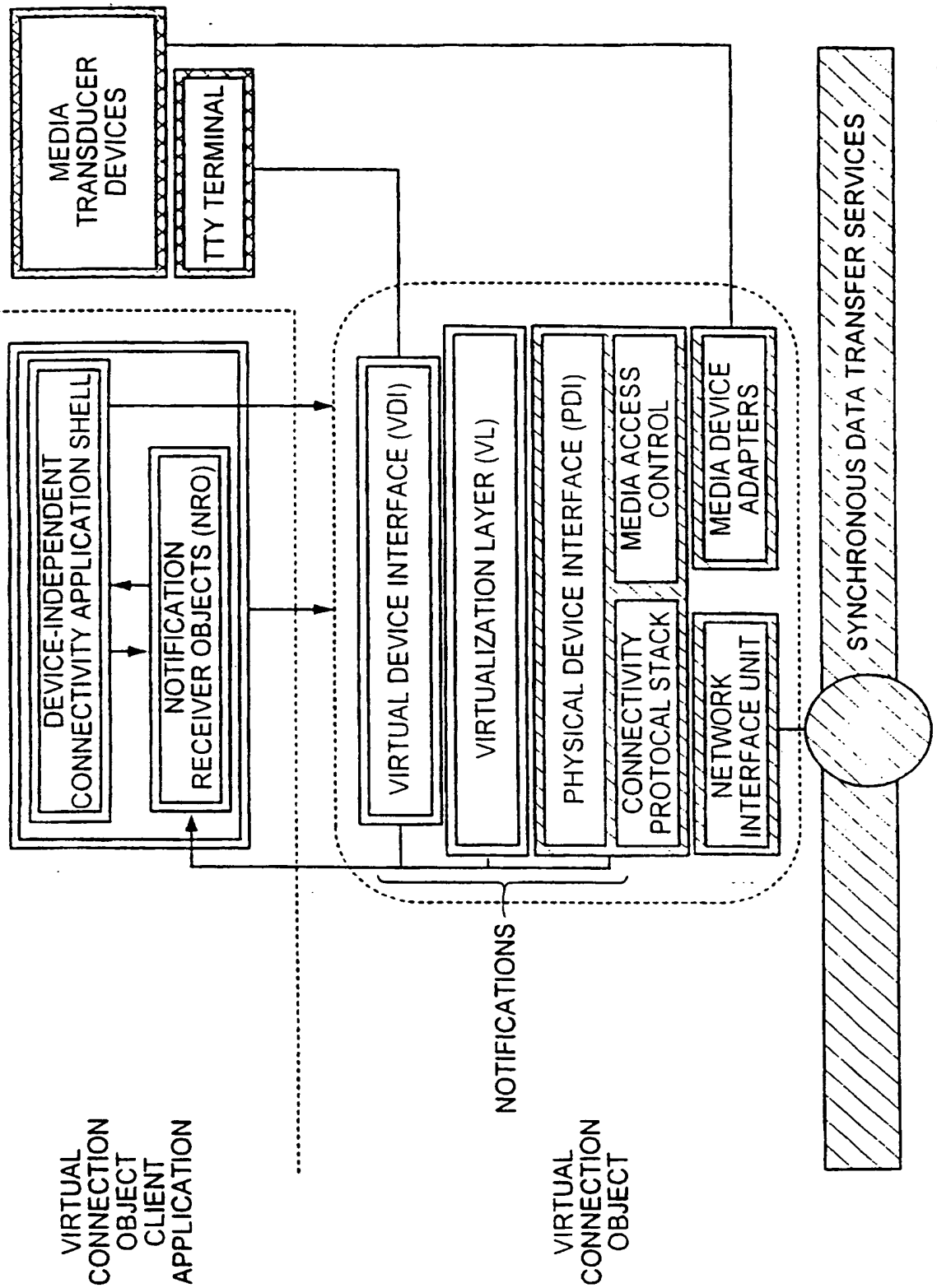


FIG. 2

FIG. 3a	FIG. 3b

FIG. 3

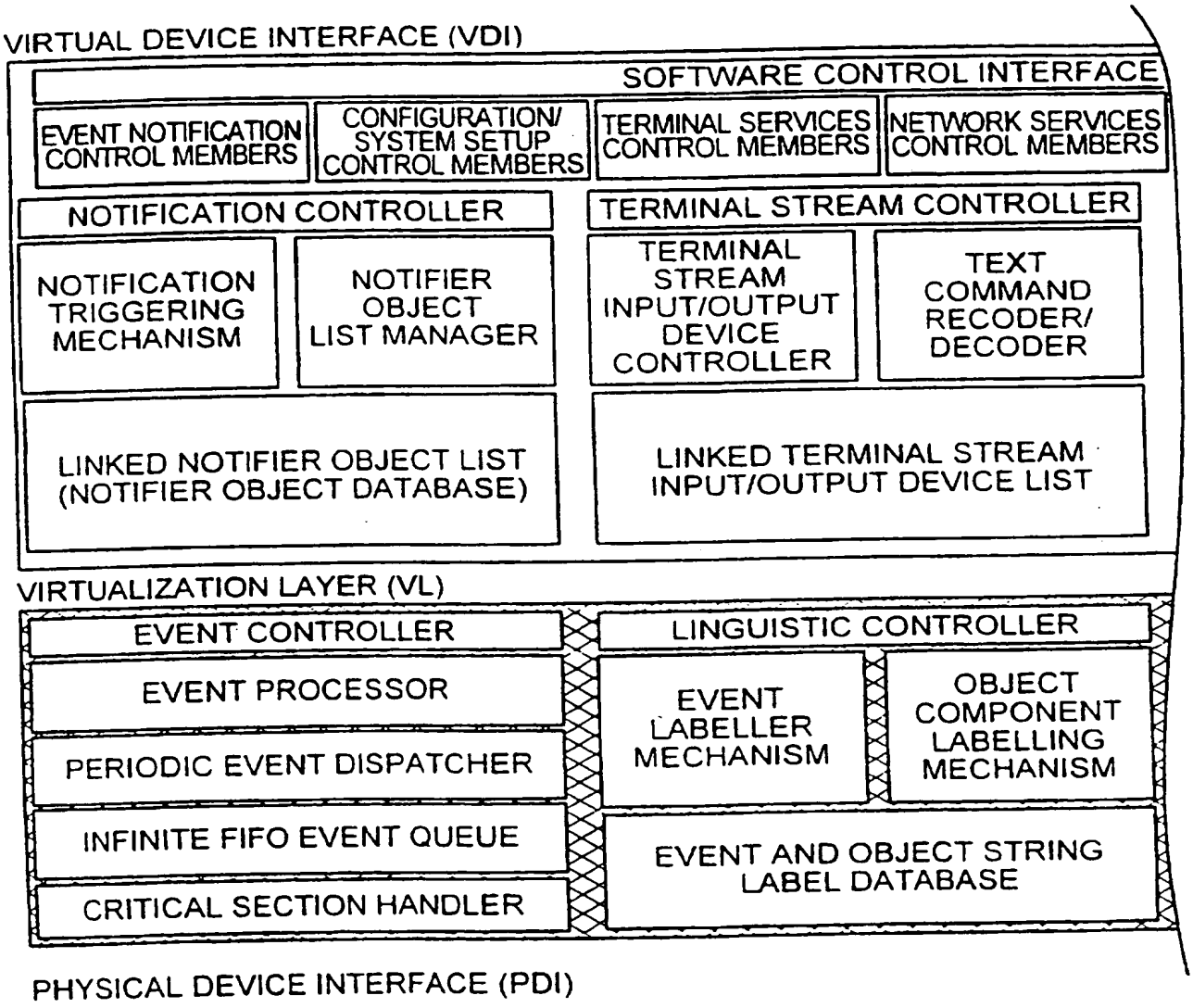


FIG. 3a

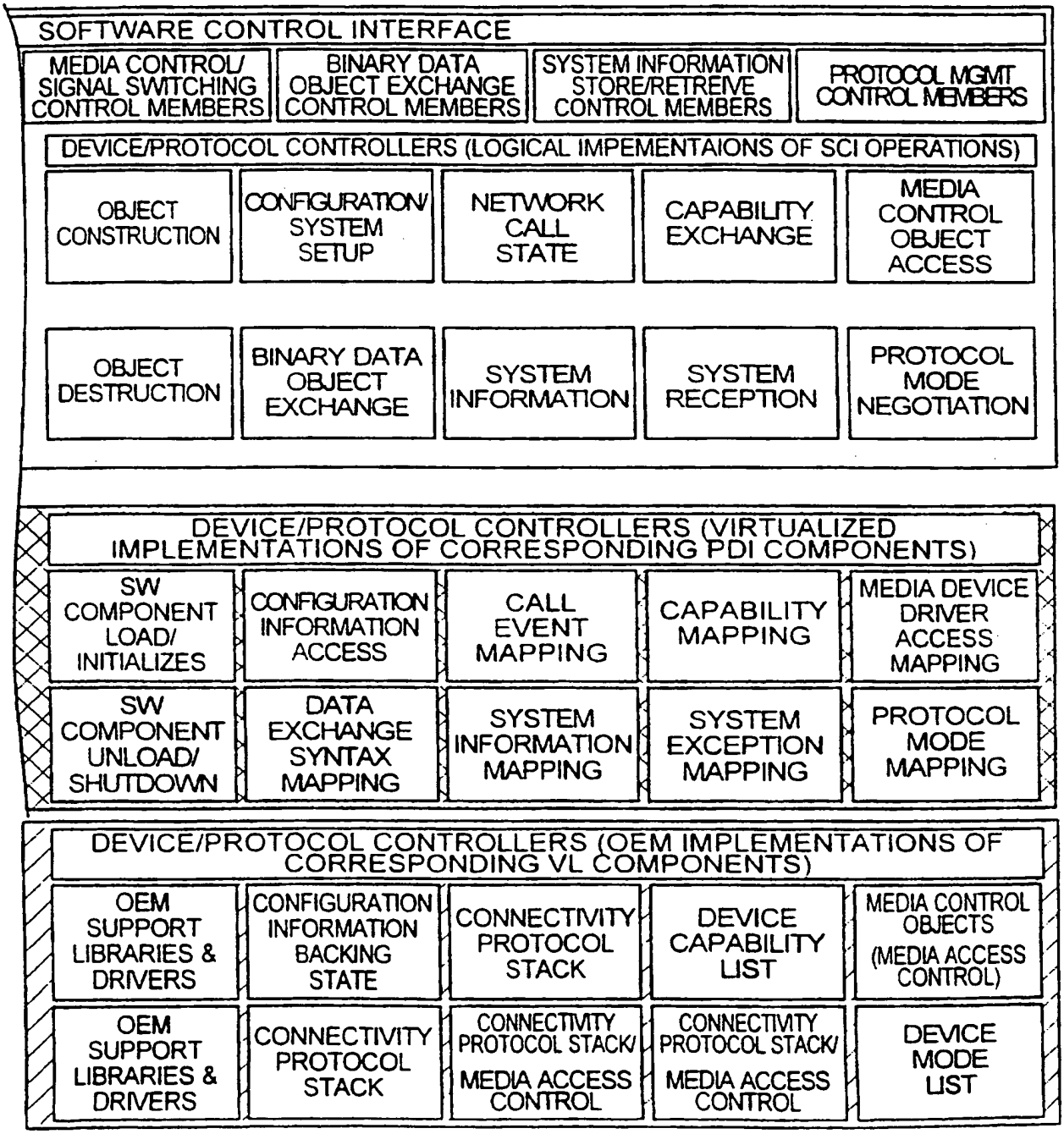

SOFTWARE CONTROL INTERFACE
MEDIA CONTROL/ SIGNAL SWITCHING CONTROL MEMBERS
BINARY DATA OBJECT EXCHANGE CONTROL MEMBERS
SYSTEM INFORMATION STORE/RETREIVE CONTROL MEMBERS
PROTOCOL MGMT CONTROL MEMBERS
DEVICE/PROTOCOL CONTROLLERS (LOGICAL IMPEMENTAIONS OF SCI OPERATIONS)
OBJECT CONSTRUCTION
CONFIGURATION/ SYSTEM SETUP
NETWORK CALL STATE
CAPABILITY EXCHANGE
MEDIA CONTROL OBJECT ACCESS
OBJECT DESTRUCTION
BINARY DATA OBJECT EXCHANGE
SYSTEM INFORMATION
SYSTEM RECEPTION
PROTOCOL MODE NEGOTIATION
DEVICE/PROTOCOL CONTROLLERS (VIRTUALIZED IMPLEMENTATIONS OF CORRESPONDING PDI COMPONENTS)
SW COMPONENT LOAD/ INITIALIZES
CONFIGURATION INFORMATION ACCESS
CALL EVENT MAPPING
CAPABILITY MAPPING
MEDIA DEVICE DRIVER ACCESS MAPPING
SW COMPONENT UNLOAD/ SHUTDOWN
DATA EXCHANGE SYNTAX MAPPING
SYSTEM INFORMATION MAPPING
SYSTEM EXCEPTION MAPPING
PROTOCOL MODE MAPPING
DEVICE/PROTOCOL CONTROLLERS (OEM IMPLEMENTATIONS OF CORRESPONDING VL COMPONENTS)
OEM SUPPORT LIBRARIES & DRIVERS
CONFIGURATION INFORMATION BACKING STATE
CONNECTIVITY PROTOCOL STACK
DEVICE CAPABILITY LIST
MEDIA CONTROL OBJECTS (MEDIA ACCESS CONTROL)
OEM SUPPORT LIBRARIES & DRIVERS
CONNECTIVITY PROTOCOL STACK
CONNECTIVITY PROTOCOL STACK/ MEDIA ACCESS CONTROL
CONNECTIVITY PROTOCOL STACK/ MEDIA ACCESS CONTROL
DEVICE MODE LIST

FIG. 3b

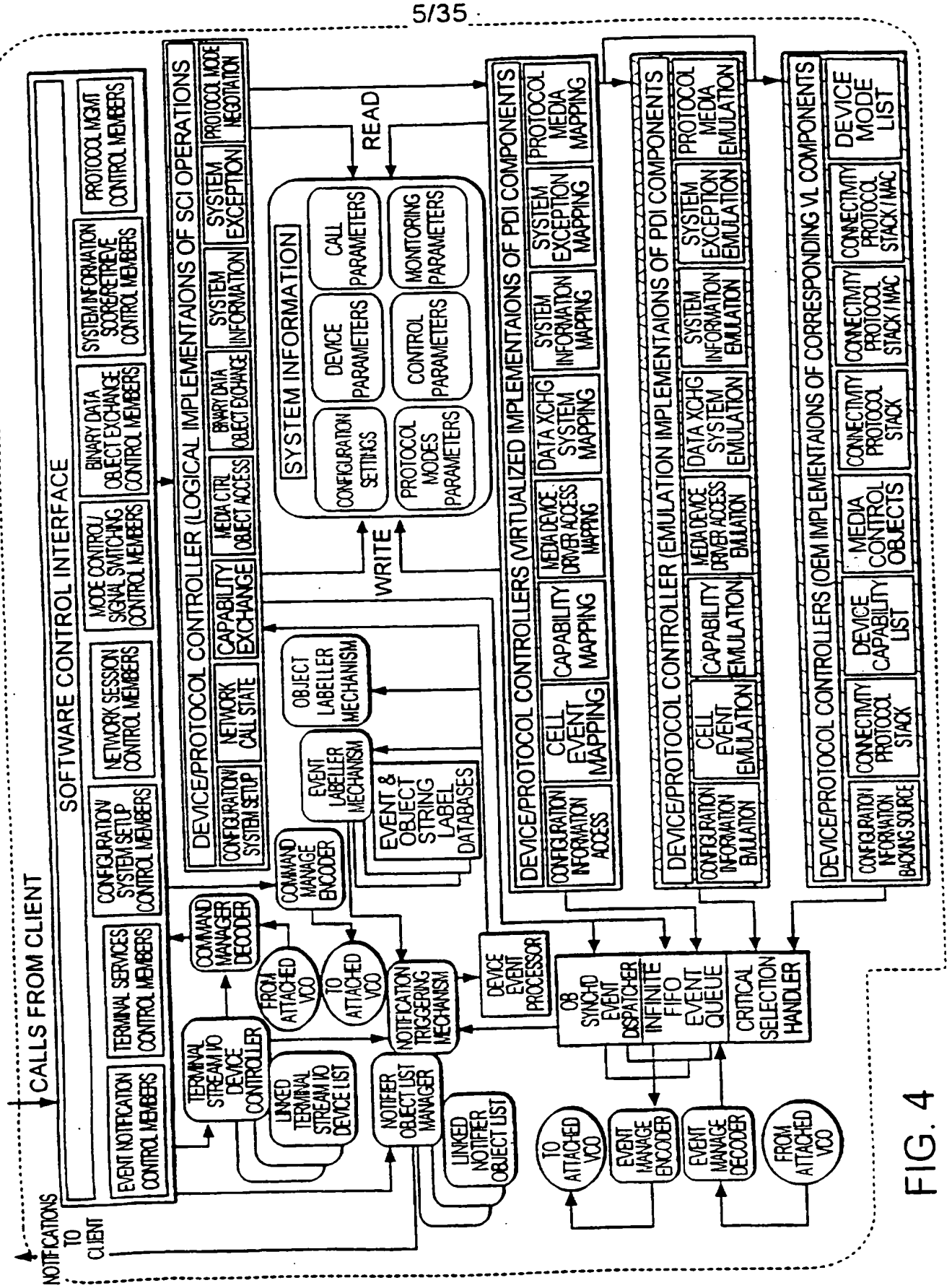

CALLS FROM CLIENT
SOFTWARE CONTROL INTERFACE
EVENT NOTIFICATION CONTROL MEMBERS
TERMINAL SERVICES CONTROL MEMBERS
CONFIGURATION SYSTEM SETUP CONTROL MEMBERS
NETWORK SESSION CONTROL MEMBERS
MODE CONTROL/ SIGNAL SWITCHING CONTROL MEMBERS
BINARY DATA OBJECT EXCHANGE CONTROL MEMBERS
SYSTEM INFORMATION SOURCE/RETRIEVE CONTROL MEMBERS
PROTOCOL MGMT CONTROL MEMBERS
NOTIFICATIONS TO CLIENT
DEVICE/PROTOCOL CONTROLLER (LOGICAL IMPLEMENTAIONS OF SCI OPERATIONS
CONFIGURATION/ SYSTEM SETUP
NETWORK CALL STATE
CAPABILITY EXCHANGE
MEDIA CTRL OBJECT ACCESS
BINARY DATA OBJECT EXCHANGE
SYSTEM INFORMATION
SYSTEM EXCEPTION
PROTOCOL MODE NEGOTIATION
READ
WRITE
SYSTEM INFORMATION
CONFIGURATION SETTINGS
DEVICE PARAMETERS
CALL PARAMETERS
PROTOCOL MODES PARAMETERS
CONTROL PARAMETERS
MONITORING PARAMETERS
COMMAND MANAGE DECODER
COMMAND MANAGE ENCODER
EVENT LABELLER MECHANISM
OBJECT LABELLER MECHANISM
EVENT & OBJECT STRING LABEL DATABASES
TERMINAL STREAM I/O DEVICE CONTROLLER
LINKED TERMINAL STREAM I/O DEVICE LIST
NOTIFIER OBJECT LIST MANAGER
LINKED NOTIFIER OBJECT LIST
FROM ATTACHED VCO
TO ATTACHED VCO
NOTIFICATION TRIGGERING MECHANISM
DEVICE EVENT PROCESSOR
DEVICE/PROTOCOL CONTROLLERS (VIRTUALIZED IMPLEMENTAIONS OF PDI COMPONENTS
CONFIGURATION INFORMATION ACCESS
CELL EVENT MAPPING
CAPABILITY MAPPING
MEDIA DEVICE DRIVER ACCESS MAPPING
DATA XCHG SYSTEM MAPPING
SYSTEM INFORMATION MAPPING
SYSTEM EXCEPTION MAPPING
PROTOCOL MEDIA MAPPING
DEVICE/PROTOCOL CONTROLLER (EMULATION IMPLEMENTAIONS OF PDI COMPONENTS
CONFIGURATION INFORMATION EMULATION
CELL EVENT EMULATION
CAPABILITY EMULATION
MEDIA DEVICE DRIVER ACCESS EMULATION
DATA XCHG SYSTEM EMULATION
SYSTEM INFORMATION EMULATION
SYSTEM EXCEPTION EMULATION
PROTOCOL MEDIA EMULATION
DEVICE/PROTOCOL CONTROLLERS (OEM IMPLEMENTAIONS OF CORRESPONDING VL COMPONENTS
CONFIGURATION INFORMATION BACKING SOURCE
CONNECTIVITY PROTOCOL STACK
DEVICE CAPABILITY LIST
MEDIA CONTROL OBJECTS
CONNECTIVITY PROTOCOL STACK
CONNECTIVITY PROTOCOL STACK / MAC
CONNECTIVITY PROTOCOL STACK / MAC
DEVICE MODE LIST
08 SYNCHD EVENT DISPATCHER
INFINITE FIFO EVENT QUEUE
CRITICAL SELECTION HANDLER
TO ATTACHED VCO
EVENT MANAGE ENCODER
EVENT MANAGE DECODER
FROM ATTACHED VCO

FIG. 4

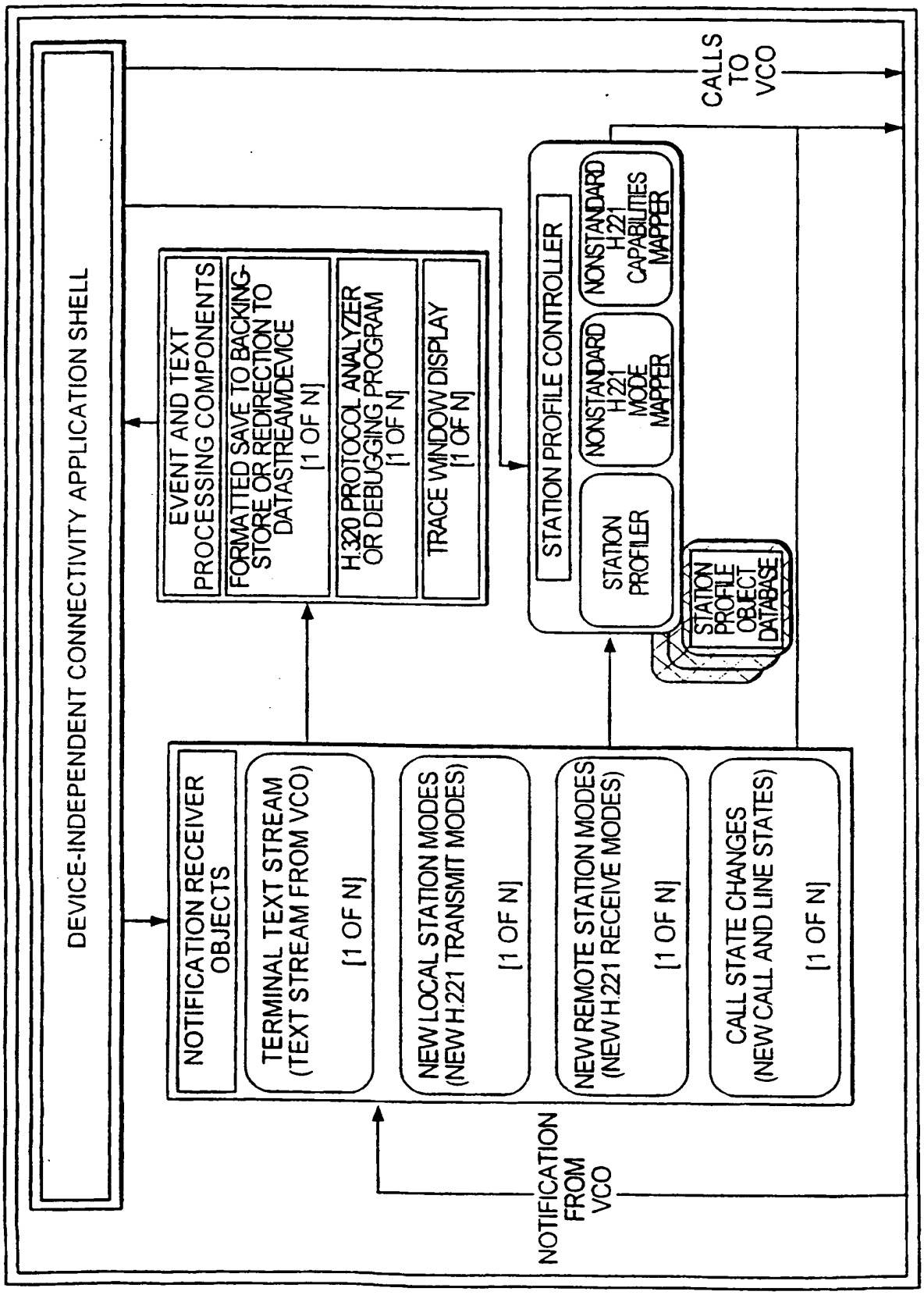

DEVICE-INDEPENDENT CONNECTIVITY APPLICATION SHELL
CALLS TO VCO
EVENT AND TEXT PROCESSING COMPONENTS
FORMATTED SAVE TO BACKING STORE OR REDIRECTION TO DATASTREAMDEVICE [1 OF N]
H.320 PROTOCOL ANALYZER OR DEBUGGING PROGRAM [1 OF N]
TRACE WINDOW DISPLAY [1 OF N]
STATION PROFILE CONTROLLER
NONSTANDARD H.221 CAPABILITIES MAPPER
NONSTANDARD H.221 MODE MAPPER
STATION PROFILER
STATION PROFILE OBJECT DATABASE
NOTIFICATION RECEIVER OBJECTS
TERMINAL TEXT STREAM (TEXT STREAM FROM VCO) [1 OF N]
NEW LOCAL STATION MODES (NEW H.221 TRANSMIT MODES) [1 OF N]
NEW REMOTE STATION MODES (NEW H.221 RECEIVE MODES) [1 OF N]
CALL STATE CHANGES (NEW CALL AND LINE STATES) [1 OF N]
NOTIFICATION FROM VCO

FIG. 5

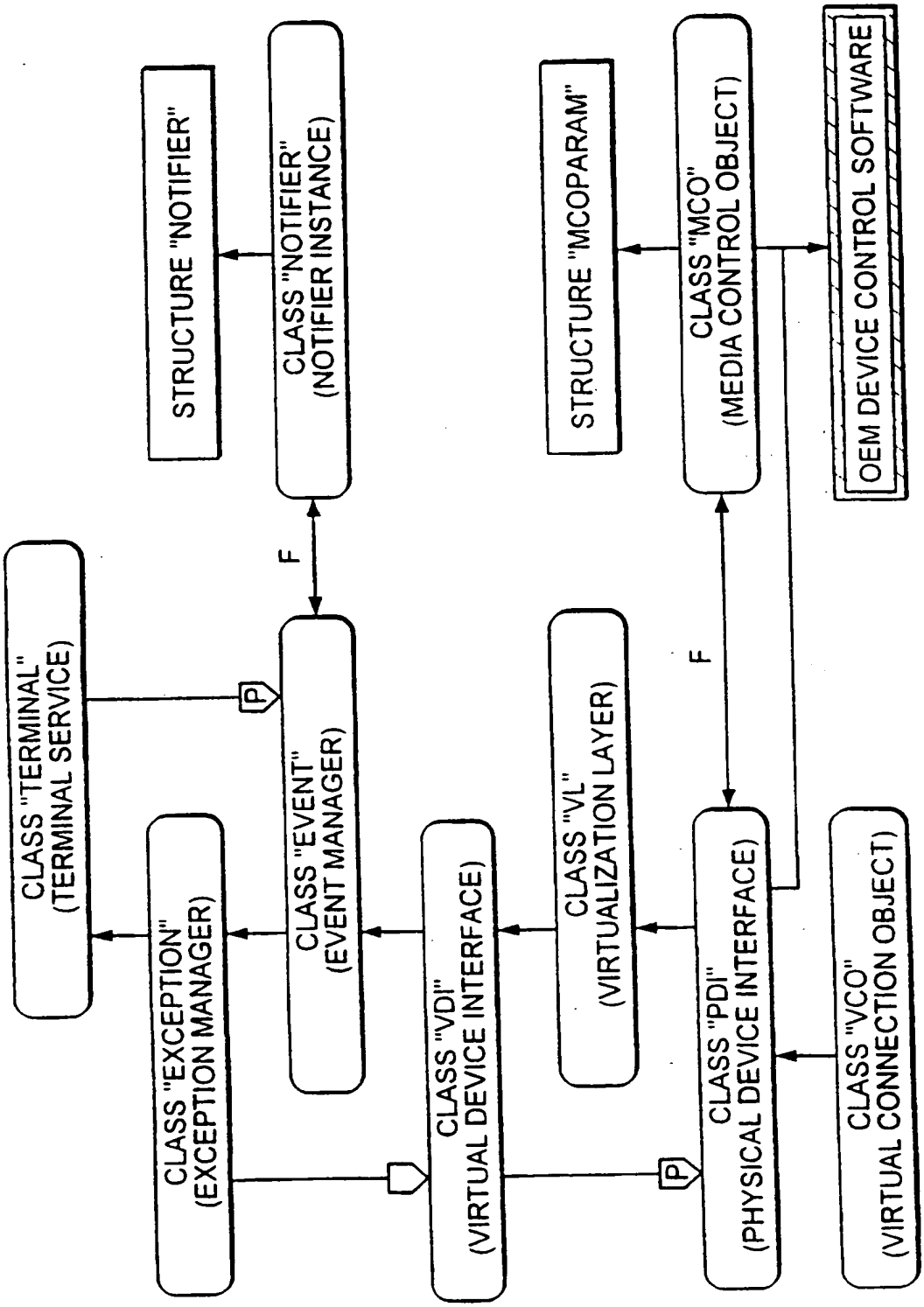

STRUCTURE "NOTIFIER"
CLASS "NOTIFIER" (NOTIFIER INSTANCE)
STRUCTURE "MCOPARAM"
CLASS "MCO" (MEDIA CONTROL OBJECT)
OEM DEVICE CONTROL SOFTWARE
F
F
CLASS "TERMINAL" (TERMINAL SERVICE)
P
CLASS "EVENT" (EVENT MANAGER)
CLASS "VL" (VIRTUALIZATION LAYER)
CLASS "EXCEPTION" (EXCEPTION MANAGER)
CLASS "VDI" (VIRTUAL DEVICE INTERFACE)
P
CLASS "PDI" (PHYSICAL DEVICE INTERFACE)
CLASS "VCO" (VIRTUAL CONNECTION OBJECT)

FIG. 6

MODE	DESCRIPTION
DEBUG	OUTPUT DEBUG INFORMATION IN MESSAGE BOX
USER	OUTPUT "USER" INFORMATION IN MESSAGE BOX
TERM	OUTPUT TEXT INFORMATION TO TERMINAL PORT
NOTIFY	REPORT EXCEPTION BY TRIGGERING SIGNAL
ABORT	ABORT CURRENT OPERATION AND DISABLE VCO

EXCEPTION HANDLING MODALITIES

FIG. 6A

MODE	DESCRIPTION
DEVICE	SEND DEVICE CONTROL MESSAGES TO TERMINAL
NOTIFY	SEND NOTIFICATION MESSAGES TO TERMINAL
MEDIA	SEND MEDIA CONTROL OBJECT MESSAGES TO TERMINAL
CALL	SEND CALL CONTOL MESSAGES TO TERMINAL
LINE	SEND LINE STATE MESSAGES TO TEMINAL
PROTOCOL	SEND PROTOCOL MESSAGE TO TERMINAL

TRACE OUTPUT MODALITIES

FIG. 6B

IDENTIFIER	SOURCE	NOTIFICATION
NULLEVENT	SW [MDI]	NO OPERATION; GENERATED BY USER AND PASSED THROUGH
NEWEMUSTATE	SW [PDI]	CHANGE IN EMULATION STATUS
NEWEMUOP	SW [MDI]	SUCCESSFUL COMPLETION OF EMULATION OPERATION
NEWREFCOUNT	SW [MDI]	CHANGE IN VCO REFERENCE COUNT
NEWDEVICEST	HW [MAC]	CHANGE IN MEDIA CONTROL DEVICE STATUS
NEWMCOFOCUS	SW [MDI]	CHANGE IN ASSIGNMENT OF DEFAULT MEDIA CONTROL OBJECTS
NEWLOCALCAPS	HW [MAC]	NEW LOCAL CAPABILITIES LIST IS AVAILABLE
NEWREMOTECAPS	HW [NIU]	NEW REMOTE CAPABILITES LIST IS AVAILABLE
NEWRCVMODE	HW [NIU]	NEW MODE HAS BEEN SET SUCCESSFULLY BY REMOTE STATION
NEWXMTMODE	HW [MAC]	NEW MODE HAS BEEN SET SUCCESSFULLY BY LOCAL STATION
NEWREJMODE	HW [NIU]	MODE CHANGE ATTEMPT BY LOCAL STATION HAS BEEN REJECTED
NEWADIOSSETTING	HW [MAC]	SUCCESSFUL ESTABLISHMENT OF NEW AUDIO OBJECT SETTING
NEWVIDEOSETTING	HW [MAC]	SUCCESSFUL ESTABLISHMENT OF NEW VIDEO OBJECT SETTING
NEWIMAGESETTING	HW [MAC]	SUCCESSFUL ESTABLISHMENT OF NEW IMAGE OBJECT SETTING
NEWDATASETTING	HW [NIU]	SUCCESSFUL ESTABLISHMENT OF NEW DATA OBJECT SETTING
NEWCALLSTATE	HW [NIU]	CHANGE IN CALL STATUS
NEWLINE1STATE	HW [NIU]	CHANGE IN LINE 1 STATUS
NEWLINE2STATE	HW [NIU]	CHANGE IN LINE 2 STATUS
NEWCONFPROFILE	SW [MDI]	NEW CONFERENCE PROFILE SET SUCCESSFULLY
NEWDISC STATUS	HW [NIU]	NEW DISCONNECTION STATUS; CALL HAS BEEN DISCONNECTED
NEWMULTICALLSTATE	HW [NIU]	CHANGE IN MULTIPOINT CALL STATUS
NEWMULTICALLOP	SW [MDI]	SUCCESSFUL COMPLETION OF MULTIPOINT OPERATION
NEWDATAXFERSTATE	SW [MDI]	NEW DATA OBJECT TRANSFER STATUS
NEWRCVBUFFER	HW [NIU]	DATA BUFFER RECEIVED FROM REMOTE STATION
NEWXMTBUFFER	HW [NIU]	DATA BUFFER SUCCESSFULLY TRANSFERRED TO REMOTE STATION
NEWRCVOBJECT	HW [NIU]	DATA OBJECT RECEIVED FROM REMOTE STATION
NEWXMTOGJECT	HW [NIU]	DATA OBJECT SUCCESSFULLY TRANSFERRED TO REMOTE STATION
NEWVCOSTATE	SW [MDI]	CHANGE IN VCO STATUS
NEWCURSORPOS	HW [NIU]	CHANGE IN GRAPHICS CURSOR POSITION
NEWTERMINPUT	SW [MDI]	TEXT MESSAGE WRITTEN TO VCO TERMINAL INPUT PORT
NEWTERMOUTPUT	SW [MDI]	TEXT MESSAGE WRITTEN TO VCO TERMINAL OUTPUT PORT
NEWRESULTCODE	SW [MDI]	SCI CALL COMPLETED; RESULT CODE FROM OPERATION AVAILABLE

FIG. 6C VCO EVENTS TRIGGERING NOTIFICATION

FIG. 7-1	FIG. 7-2

FIG. 7

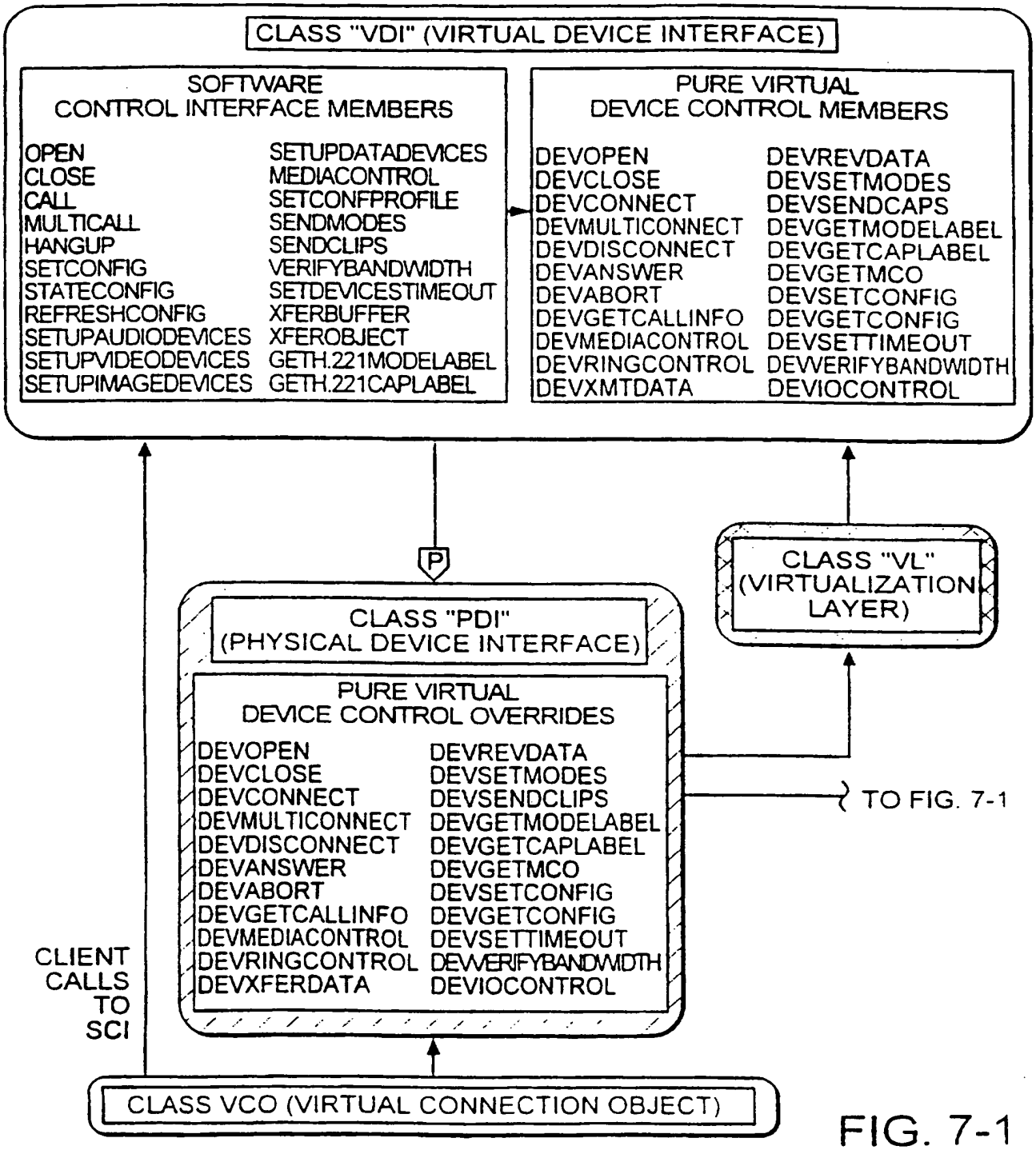


FIG. 7-1

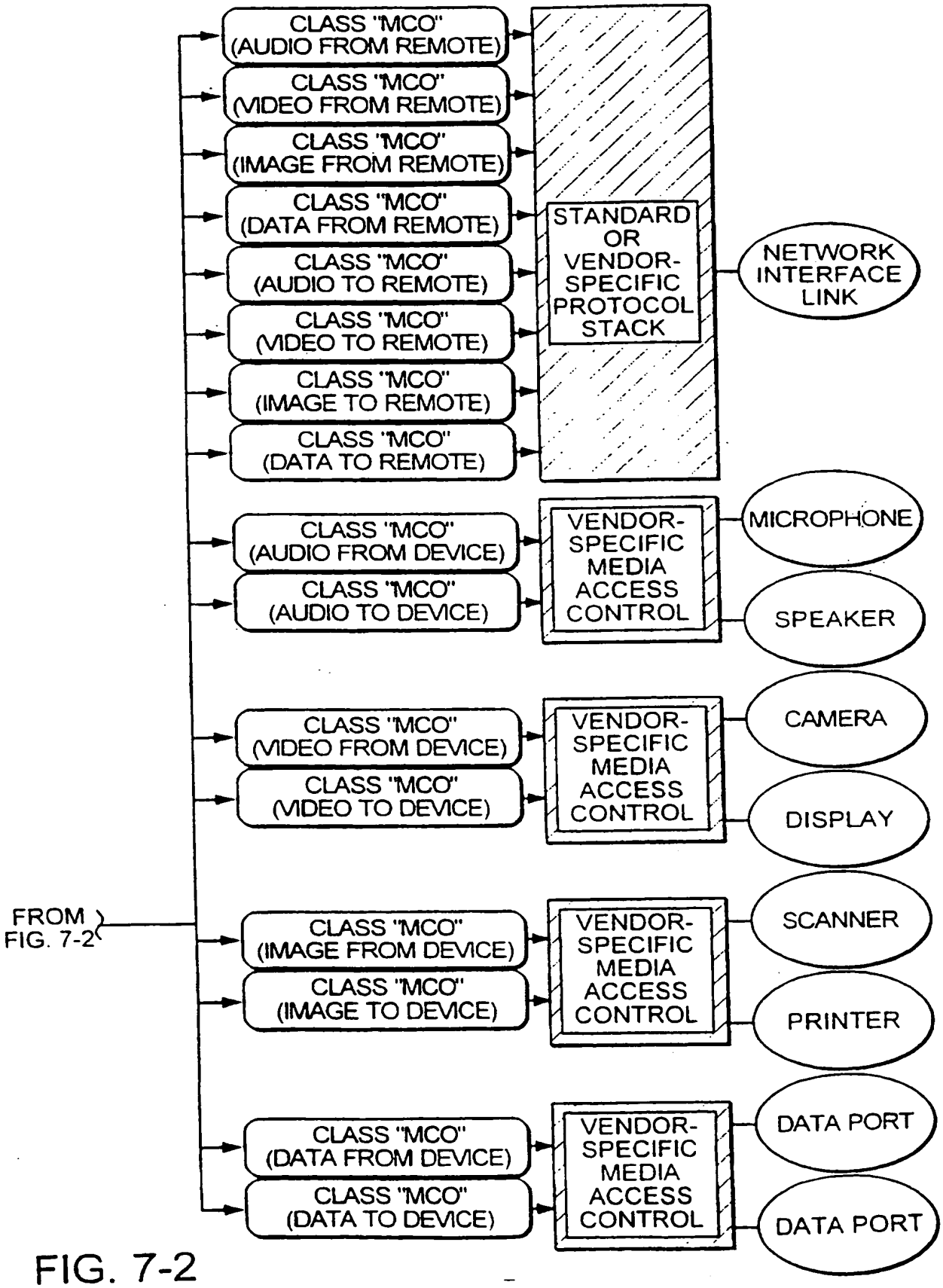

CLASS "MCO" (AUDIO FROM REMOTE)
CLASS "MCO" (VIDEO FROM REMOTE)
CLASS "MCO" (IMAGE FROM REMOTE)
CLASS "MCO" (DATA FROM REMOTE)
CLASS "MCO" (AUDIO TO REMOTE)
CLASS "MCO" (VIDEO TO REMOTE)
CLASS "MCO" (IMAGE TO REMOTE)
CLASS "MCO" (DATA TO REMOTE)
STANDARD OR VENDOR-SPECIFIC PROTOCOL STACK
NETWORK INTERFACE LINK
CLASS "MCO" (AUDIO FROM DEVICE)
CLASS "MCO" (AUDIO TO DEVICE)
VENDOR-SPECIFIC MEDIA ACCESS CONTROL
MICROPHONE
SPEAKER
CLASS "MCO" (VIDEO FROM DEVICE)
CLASS "MCO" (VIDEO TO DEVICE)
VENDOR-SPECIFIC MEDIA ACCESS CONTROL
CAMERA
DISPLAY
FROM FIG. 7-2
CLASS "MCO" (IMAGE FROM DEVICE)
CLASS "MCO" (IMAGE TO DEVICE)
VENDOR-SPECIFIC MEDIA ACCESS CONTROL
SCANNER
PRINTER
CLASS "MCO" (DATA FROM DEVICE)
CLASS "MCO" (DATA TO DEVICE)
VENDOR-SPECIFIC MEDIA ACCESS CONTROL
DATA PORT
DATA PORT

FIG. 7-2

TYPE	DESCRIPTION	DEVICE/TRANSDUCER
AUDIOIN	AUDIO SIGNAL FROM REMOTE STATION	NIU
AUDIOOUT	AUDIO SIGNAL TO REMOTE STATION	NIU
AUDIOSRC	AUDIO SIGNAL FROM INPUT TRANSDUCER	MICROPHONE/RECORDER/DISK
AUDIODST	AUDIO SIGNAL TO OUTPUT TRANSDUCER	SPEAKER/RECORDER/DISK
VIDEOIN	VIDEO SIGNAL FROM REMOTE STATION	NIU
VIDEOOUT	VIDEO SIGNAL TO REMOTE STATION	NIU
VIDEOSCR	VIDEO SIGNAL FROM INPUT TRANSDUCER	CAMERA/DISK
VIDEODST	VIDEO SIGNAL TO OUTPUT TRANSDUCER	DISPLAY/DISK
IMAGEIN	IMAGE FROM REMOTE STATION	NIU
IMAGEOUT	IMAGE TO REMOTE STATION	NIU
IMAGESRC	IMAGE FROM INPUT TRANSDUCER	SCANNER/CAMERA/DISK
IMAGEDST	IMAGE TO OUTPUT TRANSDUCER	PRINTER/DISPLAY/DISK
DATAIN	BINARY DATA STREAM FROM REMOTE STATION	NIU
DATAOUT	BINARY DATA STREAM TO REMOTE STATION	NIU
DATASCR	BINARY DATA STREAM FROM DATA PORT	COM PORT/DISK
DATADST	BINARY DATA STREAM TO DATA PORT	COM PORT/DISK

MEDIA CONTROL OBJECT TYPES

FIG. 7A

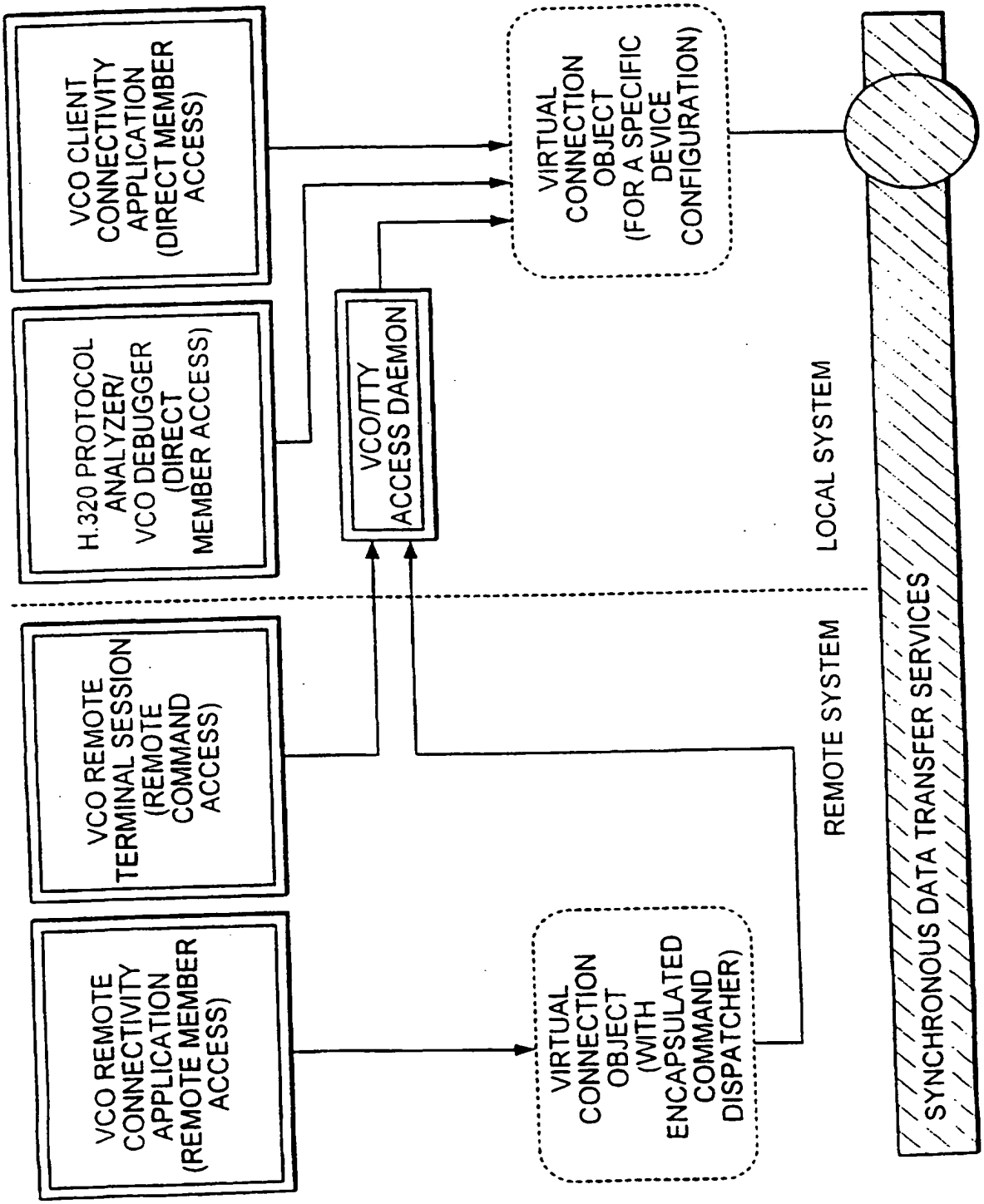
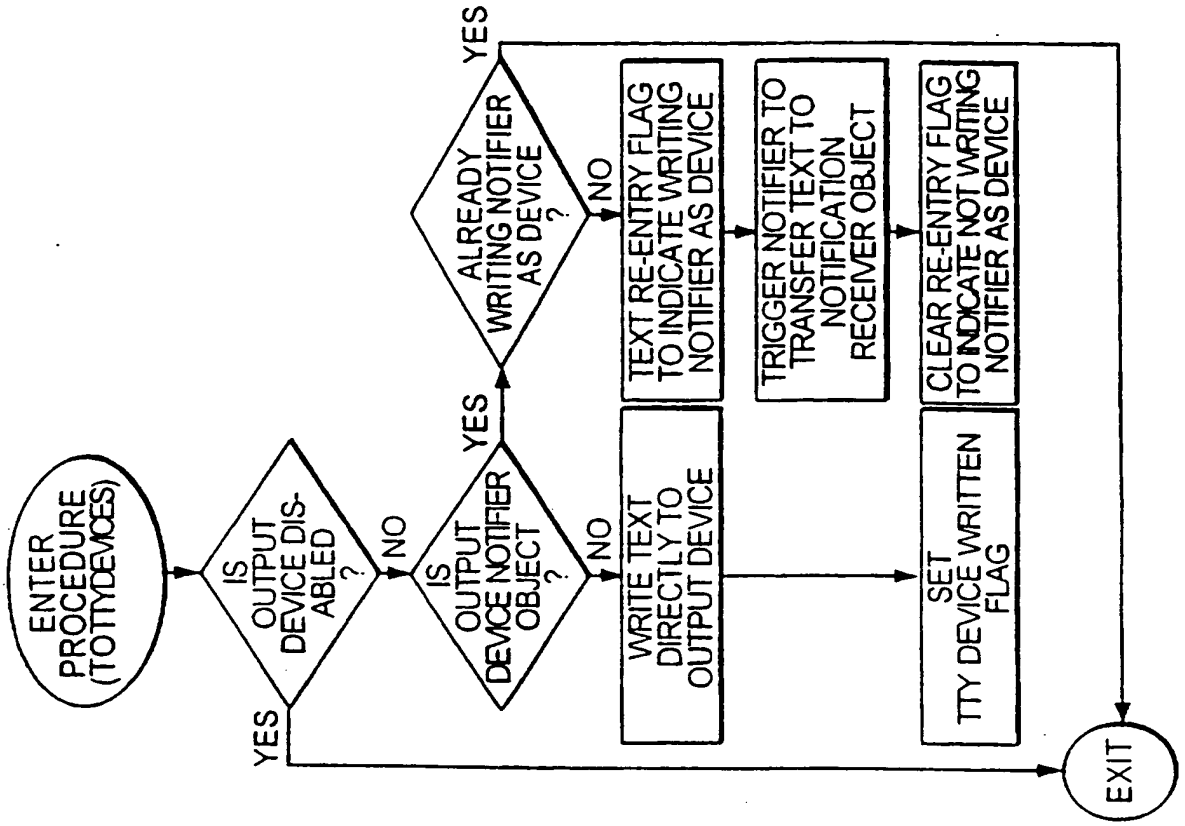

VCO CLIENT CONNECTIVITY APPLICATION (DIRECT MEMBER ACCESS)
H.320 PROTOCOL ANALYZER/ VCO DEBUGGER (DIRECT MEMBER ACCESS)
VCO REMOTE TERMINAL SESSION (REMOTE COMMAND ACCESS)
VCO REMOTE CONNECTIVITY APPLICATION (REMOTE MEMBER ACCESS)
VCO/TTY ACCESS DAEMON
VIRTUAL CONNECTION OBJECT (FOR A SPECIFIC DEVICE CONFIGURATION)
VIRTUAL CONNECTION OBJECT (WITH ENCAPSULATED COMMAND DISPATCHER)
LOCAL SYSTEM
REMOTE SYSTEM
SYNCHRONOUS DATA TRANSFER SERVICES

FIG. 8


ENTER PROCEDURE (TOTTYDEVICES)
IS OUTPUT DEVICE DIS-ABLED ?
YES
NO
IS OUTPUT DEVICE NOTIFIER OBJECT ?
YES
NO
ALREADY WRITING NOTIFIER AS DEVICE ?
YES
NO
WRITE TEXT DIRECTLY TO OUTPUT DEVICE
TEXT RE-ENTRY FLAG TO INDICATE WRITING NOTIFIER AS DEVICE
TRIGGER NOTIFIER TO TRANSFER TEXT TO NOTIFICATION RECEIVER OBJECT
CLEAR RE-ENTRY FLAG TO INDICATE NOT WRITING NOTIFIER AS DEVICE
SET TTY DEVICE WRITTEN FLAG
EXIT

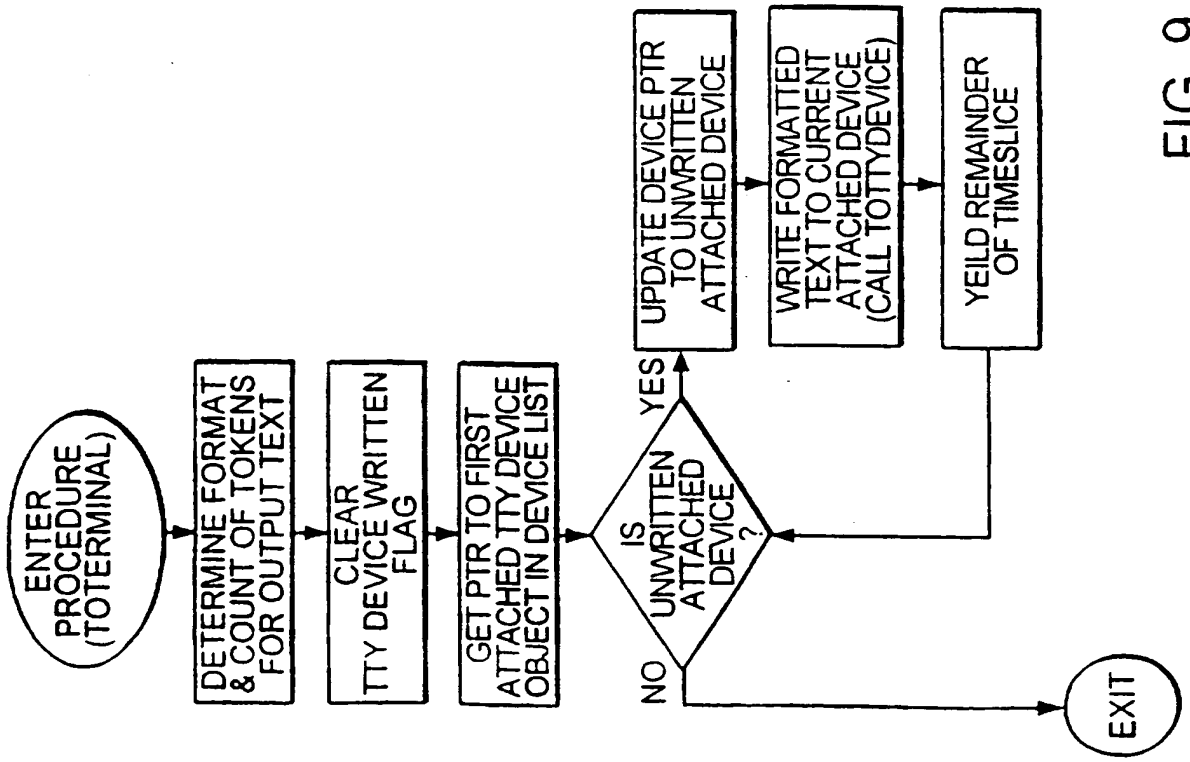

ENTER PROCEDURE (TOTERMINAL)
DETERMINE FORMAT & COUNT OF TOKENS FOR OUTPUT TEXT
CLEAR TTY DEVICE WRITTEN FLAG
GET PTR TO FIRST ATTACHED TTY DEVICE OBJECT IN DEVICE LIST
IS UNWRITTEN ATTACHED DEVICE ?
YES
NO
UPDATE DEVICE PTR TO UNWRITTEN ATTACHED DEVICE
WRITE FORMATTED TEXT TO CURRENT ATTACHED DEVICE (CALL TOTTYDEVICE)
YEILD REMAINDER OF TIMESLICE
EXIT

FIG. 9

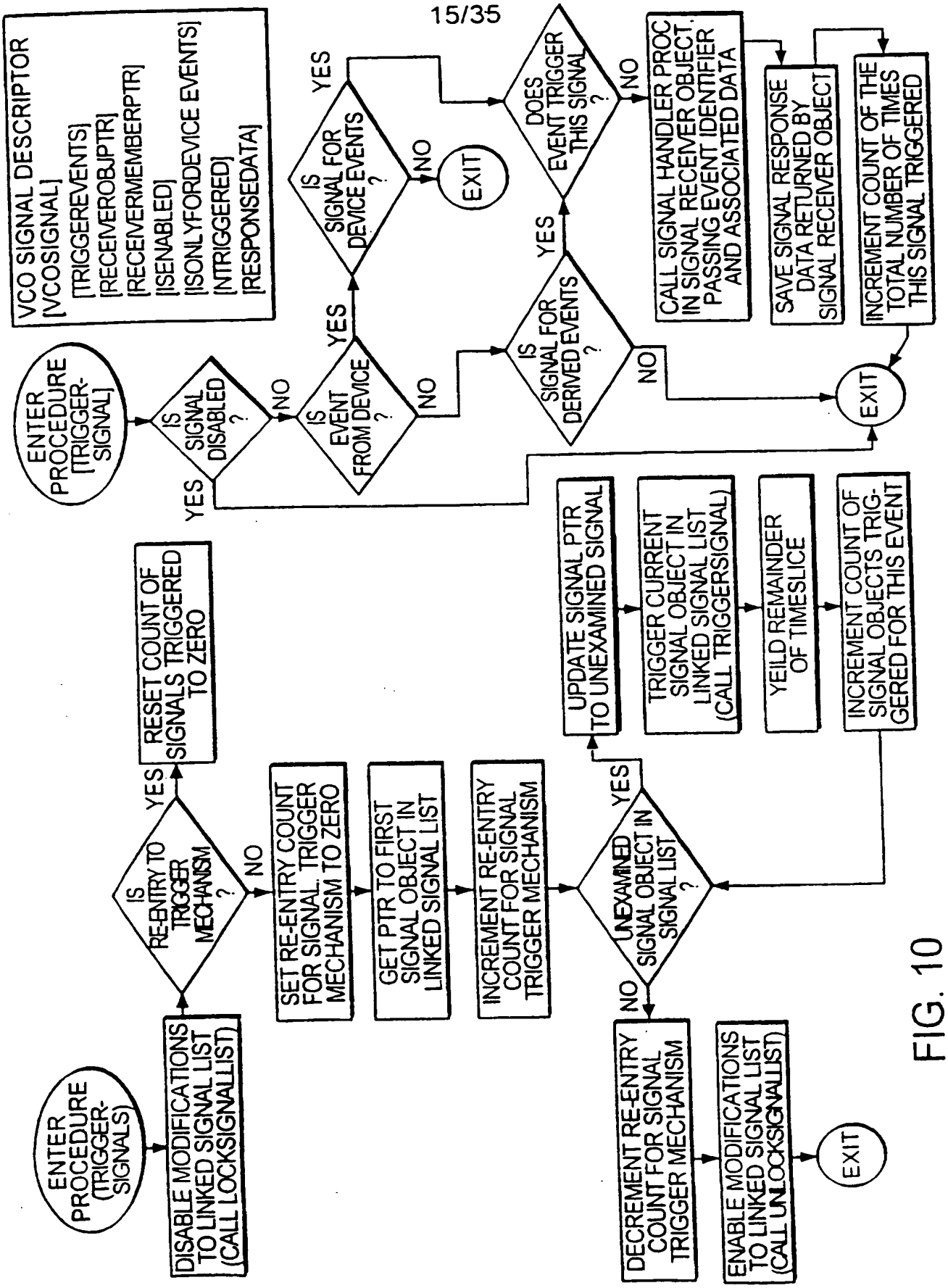

VCO SIGNAL DESCRIPTOR
[VCOSIGNAL]
[TRIGGEREVENTS]
[RECEIVEROBJPTR]
[RECEIVERMEMBERPTR]
[ISENABLED]
[ISONLYFORDEVICE EVENTS]
[NTRIGGERED]
[RESPONSEDATA]
ENTER PROCEDURE [TRIGGER-SIGNAL]
IS SIGNAL DISABLED ?
YES
NO
IS EVENT FROM DEVICE ?
YES
NO
IS SIGNAL FOR DEVICE EVENTS ?
YES
NO
EXIT
IS SIGNAL FOR DERIVED EVENTS ?
YES
NO
DOES EVENT TRIGGER THIS SIGNAL ?
YES
NO
CALL SIGNAL HANDLER PROC IN SIGNAL RECEIVER OBJECT PASSING EVENT IDENTIFIER AND ASSOCIATED DATA
SAVE SIGNAL RESPONSE DATA RETURNED BY SIGNAL RECEIVER OBJECT
INCREMENT COUNT OF THE TOTAL NUMBER OF TIMES THIS SIGNAL TRIGGERED
EXIT
ENTER PROCEDURE (TRIGGER-SIGNALS)
DISABLE MODIFICATIONS TO LINKED SIGNAL LIST (CALL LOCKSIGNALLIST)
IS RE-ENTRY TO TRIGGER MECHANISM ?
YES
NO
RESET COUNT OF SIGNALS TRIGGERED TO ZERO
SET RE-ENTRY COUNT FOR SIGNAL TRIGGER MECHANISM TO ZERO
GET PTR TO FIRST SIGNAL OBJECT IN LINKED SIGNAL LIST
INCREMENT RE-ENTRY COUNT FOR SIGNAL TRIGGER MECHANISM
UNEXAMINED SIGNAL OBJECT IN SIGNAL LIST ?
YES
NO
UPDATE SIGNAL PTR TO UNEXAMINED SIGNAL
TRIGGER CURRENT SIGNAL OBJECT IN LINKED SIGNAL LIST (CALL TRIGGERSIGNAL)
YEILD REMAINDER OF TIMESLICE
INCREMENT COUNT OF SIGNAL OBJECTS TRIG-GERED FOR THIS EVENT
DECREMENT RE-ENTRY COUNT FOR SIGNAL TRIGGER MECHANISM
ENABLE MODIFICATIONS TO LINKED SIGNAL LIST (CALL UNLOCKSIGNALLIST)
EXIT

FIG. 10

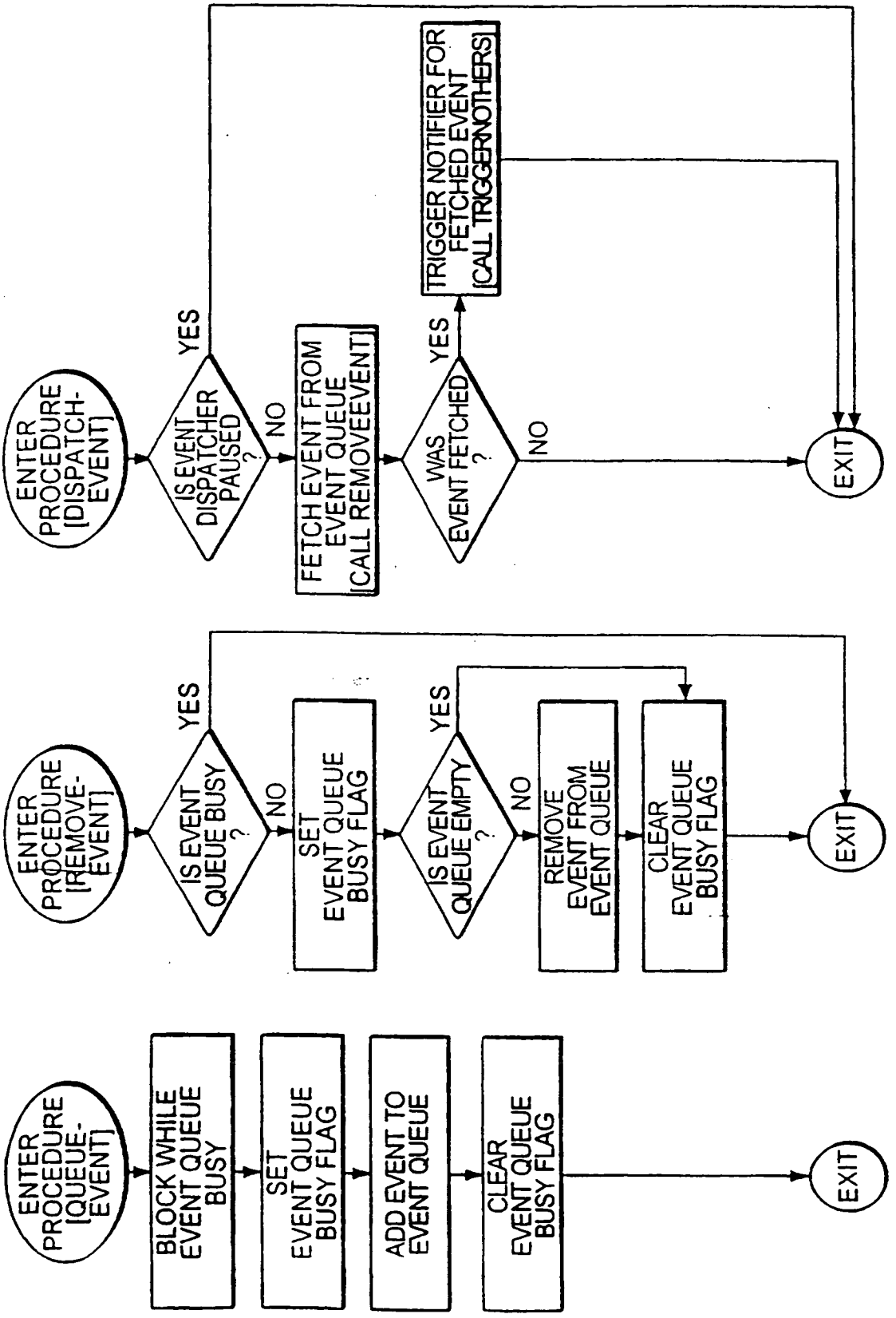

ENTER PROCEDURE [DISPATCH-EVENT]
IS EVENT DISPATCHER PAUSED ?
YES
NO
FETCH EVENT FROM EVENT QUEUE [CALL REMOVEEVENT]
WAS EVENT FETCHED ?
YES
NO
TRIGGER NOTIFIER FOR FETCHED EVENT [CALL TRIGGERNOTHERS]
EXIT
ENTER PROCEDURE [REMOVE-EVENT]
IS EVENT QUEUE BUSY ?
YES
NO
SET EVENT QUEUE BUSY FLAG
IS EVENT QUEUE EMPTY ?
YES
NO
REMOVE EVENT FROM EVENT QUEUE
CLEAR EVENT QUEUE BUSY FLAG
EXIT
ENTER PROCEDURE [QUEUE-EVENT]
BLOCK WHILE EVENT QUEUE BUSY
SET EVENT QUEUE BUSY FLAG
ADD EVENT TO EVENT QUEUE
CLEAR EVENT QUEUE BUSY FLAG
EXIT

FIG. 11

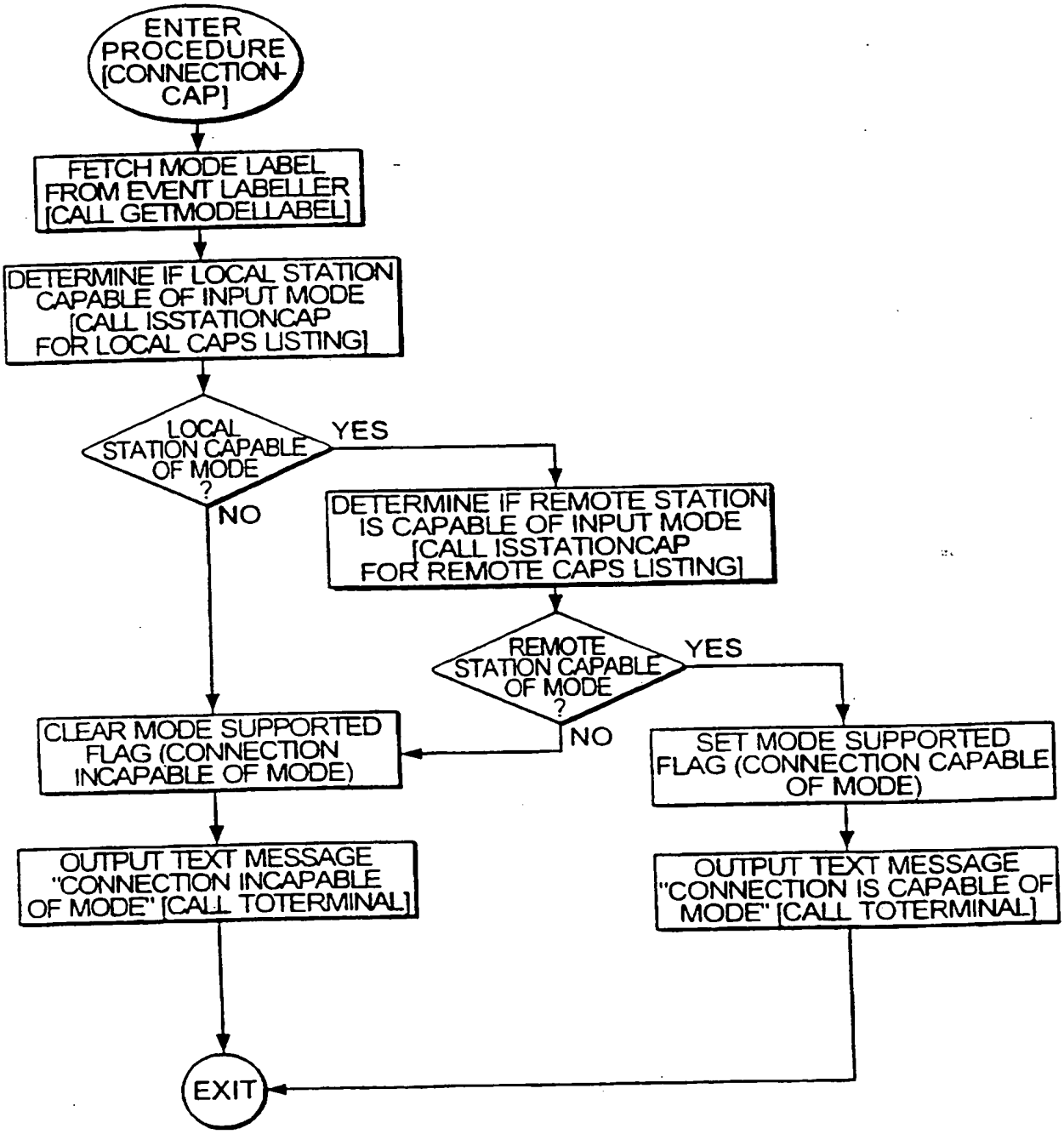

ENTER PROCEDURE [CONNECTION-CAP]
FETCH MODE LABEL FROM EVENT LABELLER [CALL GETMODELLABEL]
DETERMINE IF LOCAL STATION CAPABLE OF INPUT MODE [CALL ISSTATIONCAP FOR LOCAL CAPS LISTING]
LOCAL STATION CAPABLE OF MODE ?
YES
NO
DETERMINE IF REMOTE STATION IS CAPABLE OF INPUT MODE [CALL ISSTATIONCAP FOR REMOTE CAPS LISTING]
REMOTE STATION CAPABLE OF MODE ?
YES
NO
CLEAR MODE SUPPORTED FLAG (CONNECTION INCAPABLE OF MODE)
SET MODE SUPPORTED FLAG (CONNECTION CAPABLE OF MODE)
OUTPUT TEXT MESSAGE "CONNECTION INCAPABLE OF MODE" [CALL TOTERMINAL]
OUTPUT TEXT MESSAGE "CONNECTION IS CAPABLE OF MODE" [CALL TOTERMINAL]
EXIT

FIG. 12

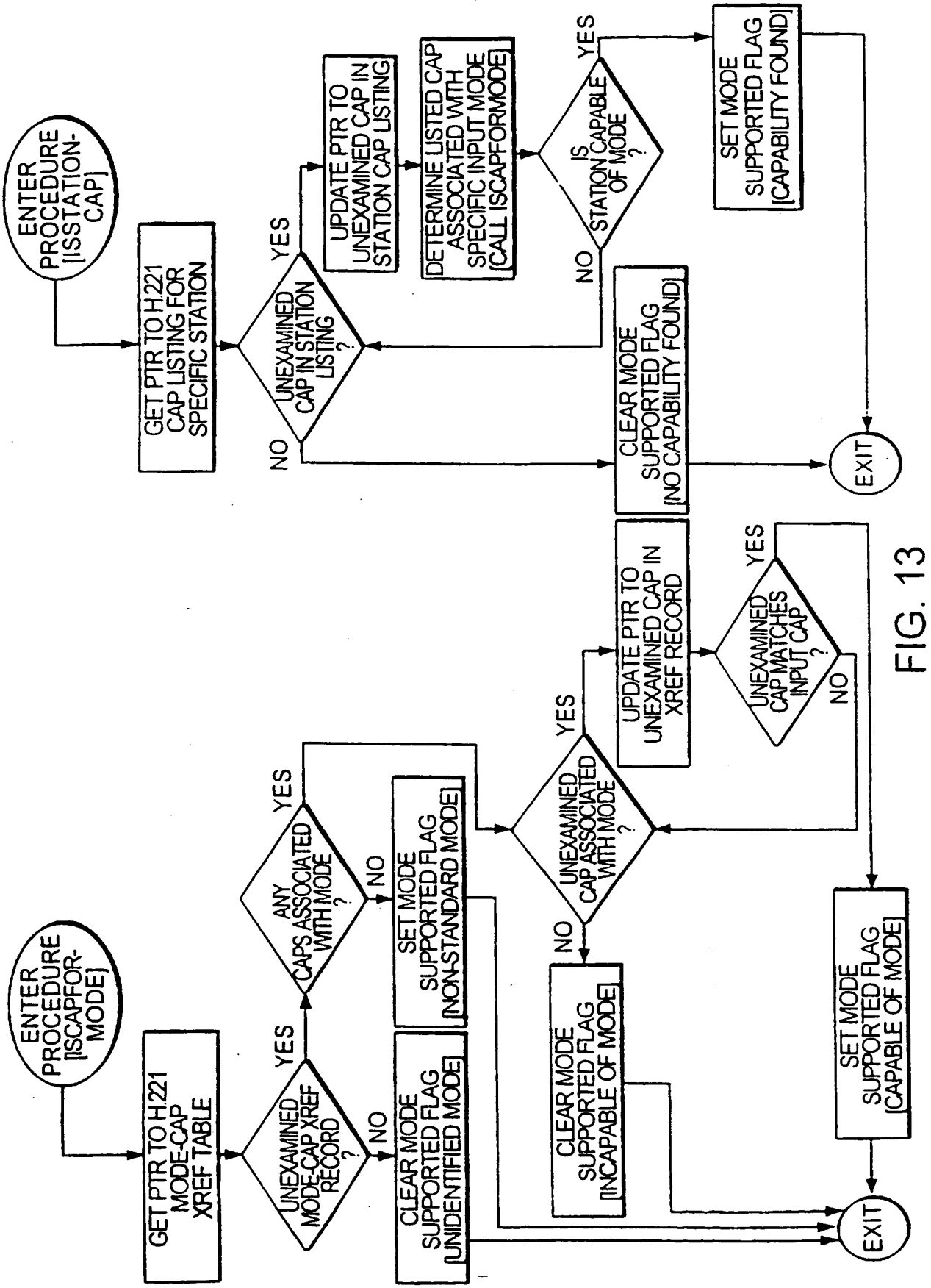
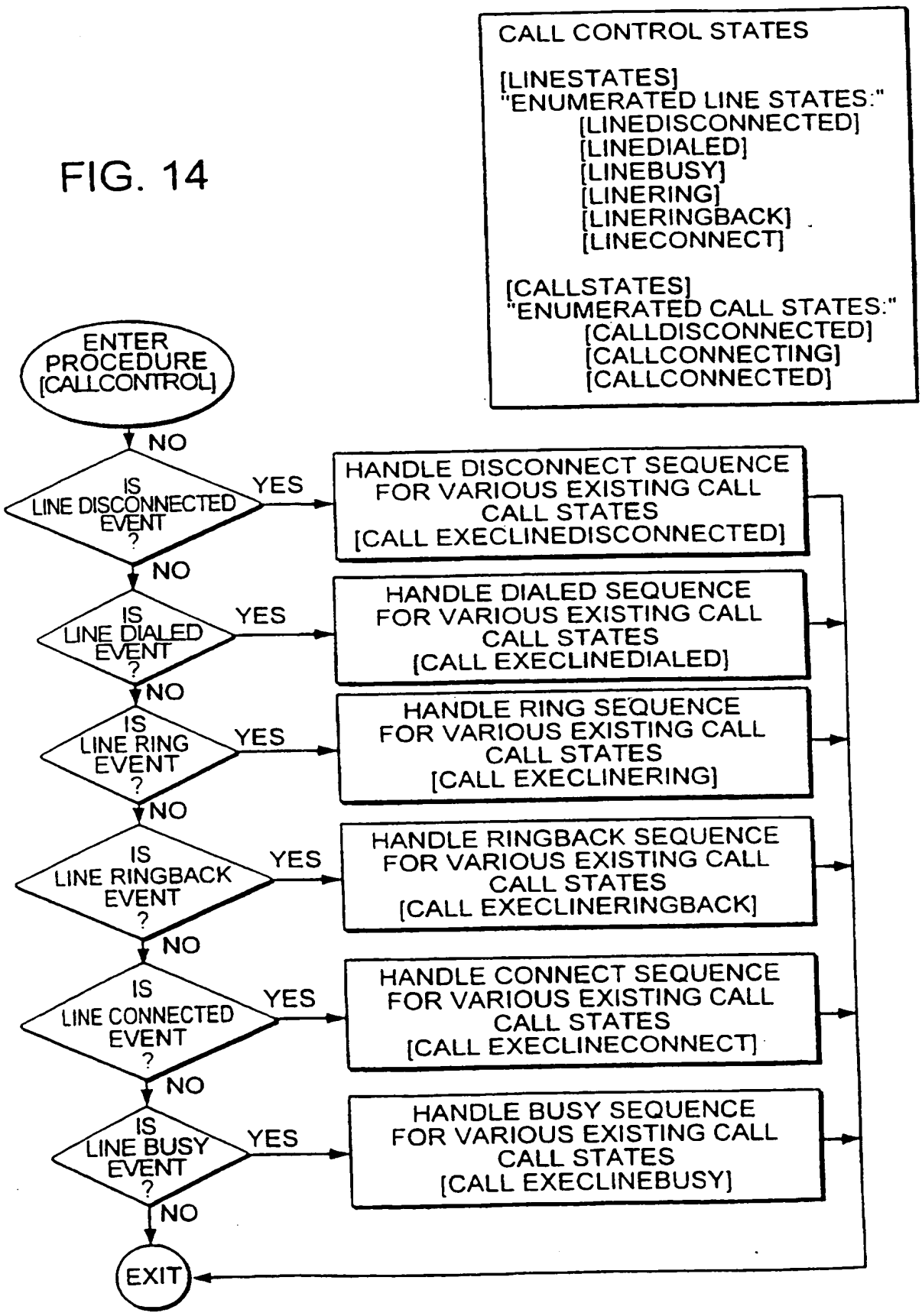

ENTER PROCEDURE [ISSTATION-CAP]
GET PTR TO H.221 CAP LISTING FOR SPECIFIC STATION
UNEXAMINED CAP IN STATION LISTING ?
YES
NO
UPDATE PTR TO UNEXAMINED CAP IN STATION CAP LISTING
DETERMINE LISTED CAP ASSOCIATED WITH SPECIFIC INPUT MODE [CALL ISCAPFORMODE]
IS STATION CAPABLE OF MODE ?
YES
NO
SET MODE SUPPORTED FLAG [CAPABILITY FOUND]
CLEAR MODE SUPPORTED FLAG [NO CAPABILITY FOUND]
EXIT
ENTER PROCEDURE [ISCAPFOR-MODE]
GET PTR TO H.221 MODE-CAP XREF TABLE
UNEXAMINED MODE-CAP XREF RECORD ?
YES
NO
CLEAR MODE SUPPORTED FLAG [UNIDENTIFIED MODE]
ANY CAPS ASSOCIATED WITH MODE ?
YES
NO
SET MODE SUPPORTED FLAG [NON-STANDARD MODE]
UNEXAMINED CAP ASSOCIATED WITH MODE ?
YES
NO
UPDATE PTR TO UNEXAMINED CAP IN XREF RECORD
UNEXAMINED CAP MATCHES INPUT CAP ?
YES
NO
CLEAR MODE SUPPORTED FLAG [INCAPABLE OF MODE]
SET MODE SUPPORTED FLAG [CAPABLE OF MODE]
EXIT

FIG. 13

FIG. 14


CALL CONTROL STATES
[LINESTATES]
"ENUMERATED LINE STATES:"
[LINEDISCONNECTED]
[LINEDIALED]
[LINEBUSY]
[LINERING]
[LINERINGBACK]
[LINECONNECT]
[CALLSTATES]
"ENUMERATED CALL STATES:"
[CALLDISCONNECTED]
[CALLCONNECTING]
[CALLCONNECTED]
ENTER PROCEDURE [CALLCONTROL]
IS LINE DISCONNECTED EVENT ?
YES
HANDLE DISCONNECT SEQUENCE FOR VARIOUS EXISTING CALL CALL STATES [CALL EXECLINEDISCONNECTED]
NO
IS LINE DIALED EVENT ?
YES
HANDLE DIALED SEQUENCE FOR VARIOUS EXISTING CALL CALL STATES [CALL EXECLINEDIALED]
NO
IS LINE RING EVENT ?
YES
HANDLE RING SEQUENCE FOR VARIOUS EXISTING CALL CALL STATES [CALL EXECLINERING]
NO
IS LINE RINGBACK EVENT ?
YES
HANDLE RINGBACK SEQUENCE FOR VARIOUS EXISTING CALL CALL STATES [CALL EXECLINERINGBACK]
NO
IS LINE CONNECTED EVENT ?
YES
HANDLE CONNECT SEQUENCE FOR VARIOUS EXISTING CALL CALL STATES [CALL EXECLINECONNECT]
NO
IS LINE BUSY EVENT ?
YES
HANDLE BUSY SEQUENCE FOR VARIOUS EXISTING CALL CALL STATES [CALL EXECLINEBUSY]
NO
EXIT

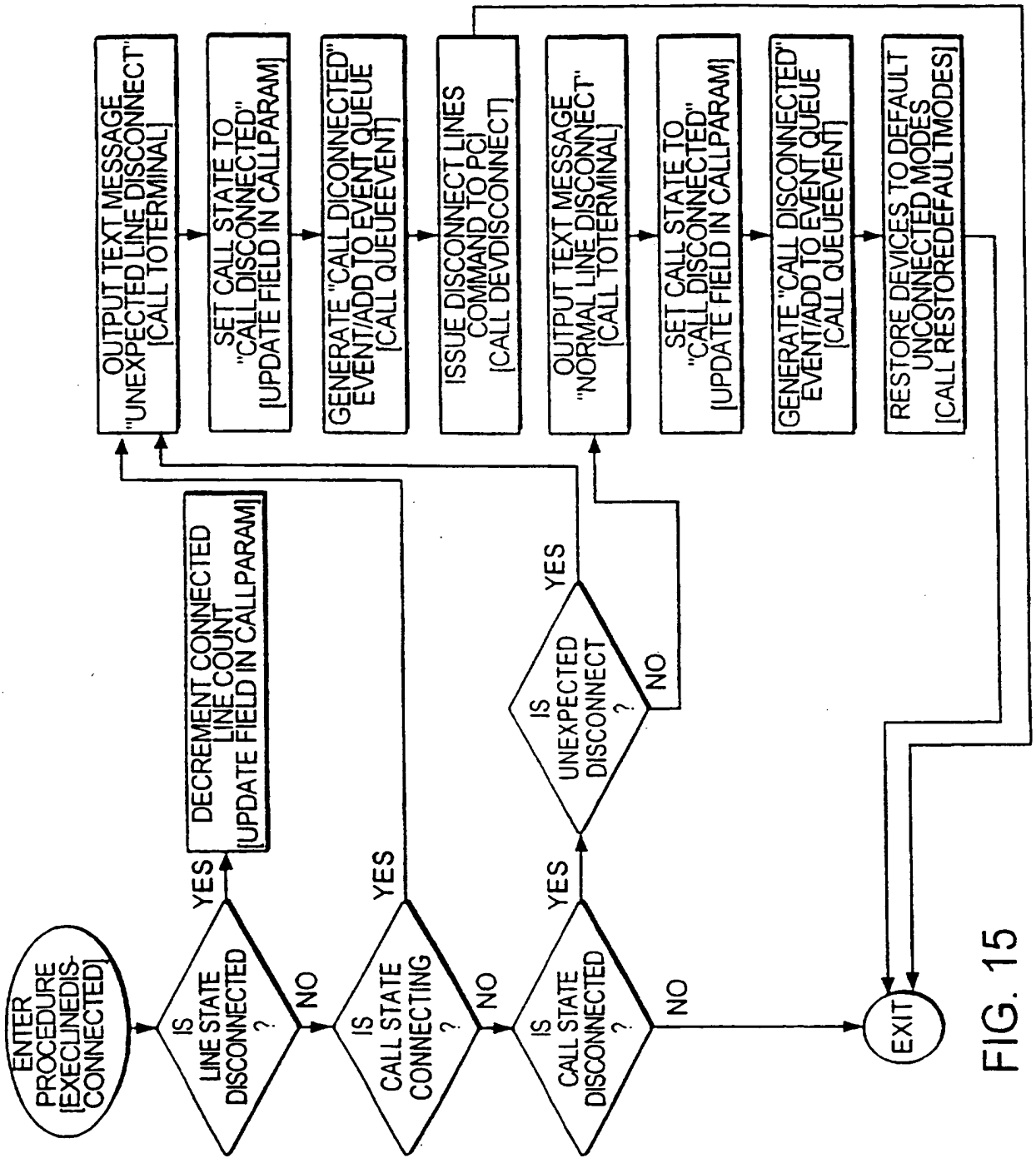

ENTER PROCEDURE [EXECLINEDIS-CONNECTED]
IS LINE STATE DISCONNECTED ?
YES
NO
DECREMENT CONNECTED LINE COUNT [UPDATE FIELD IN CALLPARAM]
IS CALL STATE CONNECTING ?
YES
NO
IS CALL STATE DISCONNECTED ?
YES
NO
IS UNEXPECTED DISCONNECT ?
YES
NO
OUTPUT TEXT MESSAGE "UNEXPECTED LINE DISCONNECT" [CALL TOTERMINAL]
SET CALL STATE TO "CALL DISCONNECTED" [UPDATE FIELD IN CALLPARAM]
GENERATE "CALL DICONNECTED" EVENT/ADD TO EVENT QUEUE [CALL QUEUEEVENT]
ISSUE DISCONNECT LINES COMMAND TO PCI [CALL DEVDISCONNECT]
OUTPUT TEXT MESSAGE "NORMAL LINE DISCONNECT" [CALL TOTERMINAL]
SET CALL STATE TO "CALL DISCONNECTED" [UPDATE FIELD IN CALLPARAM]
GENERATE "CALL DISCONNECTED" EVENT/ADD TO EVENT QUEUE [CALL QUEUEEVENT]
RESTORE DEVICES TO DEFAULT UNCONNECTED MODES [CALL RESTOREDEFAULTMODES]
EXIT

FIG. 15

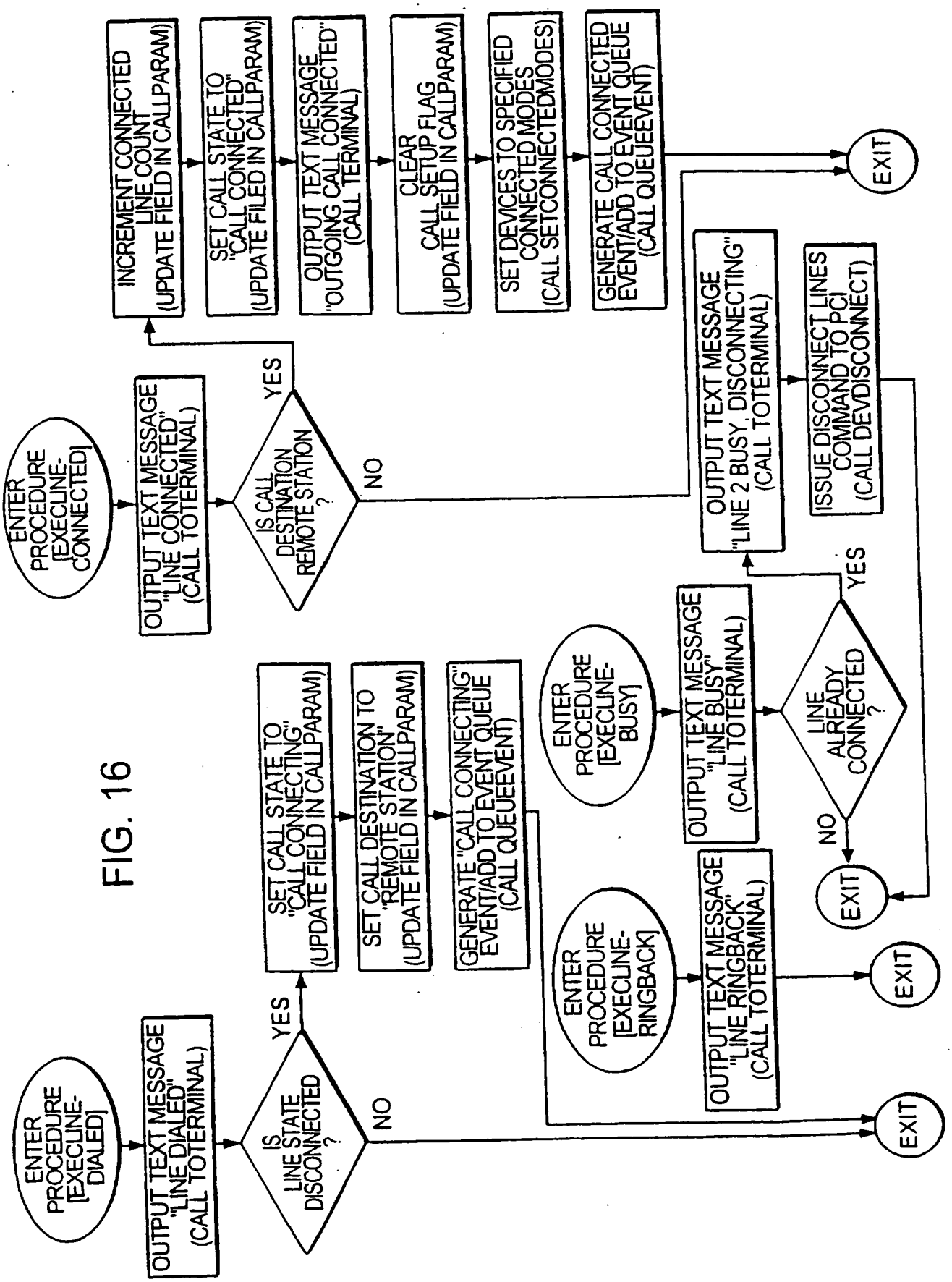

ENTER PROCEDURE [EXECUTE-DIALED]
OUTPUT TEXT MESSAGE "LINE DIALED" (CALL TOTERMINAL)
IS LINE STATE DISCONNECTED ?
YES
NO
SET CALL STATE TO "CALL CONNECTING" (UPDATE FIELD IN CALLPARAM)
SET CALL DESTINATION TO "REMOTE STATION" (UPDATE FIELD IN CALLPARAM)
GENERATE "CALL CONNECTING" EVENT/ADD TO EVENT QUEUE (CALL QUEUEEVENT)
ENTER PROCEDURE [EXECUTE-CONNECTED]
OUTPUT TEXT MESSAGE "LINE CONNECTED" (CALL TOTERMINAL)
IS CALL DESTINATION REMOTE STATION ?
YES
NO
INCREMENT CONNECTED LINE COUNT (UPDATE FIELD IN CALLPARAM)
SET CALL STATE TO "CALL CONNECTED" (UPDATE FILED IN CALLPARAM)
OUTPUT TEXT MESSAGE "OUTGOING CALL CONNECTED" (CALL TERMINAL)
CLEAR CALL SETUP FLAG (UPDATE FIELD IN CALLPARAM)
SET DEVICES TO SPECIFIED CONNECTED MODES (CALL SETCONNECTEDMODES)
GENERATE CALL CONNECTED EVENT/ADD TO EVENT QUEUE (CALL QUEUEEVENT)
EXIT
ENTER PROCEDURE [EXECUTE-BUSY]
OUTPUT TEXT MESSAGE "LINE BUSY" (CALL TOTERMINAL)
LINE ALREADY CONNECTED ?
YES
NO
OUTPUT TEXT MESSAGE "LINE 2 BUSY, DISCONNECTING" (CALL TOTERMINAL)
ISSUE DISCONNECT LINES COMMAND TO PCI (CALL DEVDISCONNECT)
EXIT
ENTER PROCEDURE [EXECUTE-RINGBACK]
OUTPUT TEXT MESSAGE "LINE RINGBACK" (CALL TOTERMINAL)
EXIT
EXIT
EXIT

FIG. 16

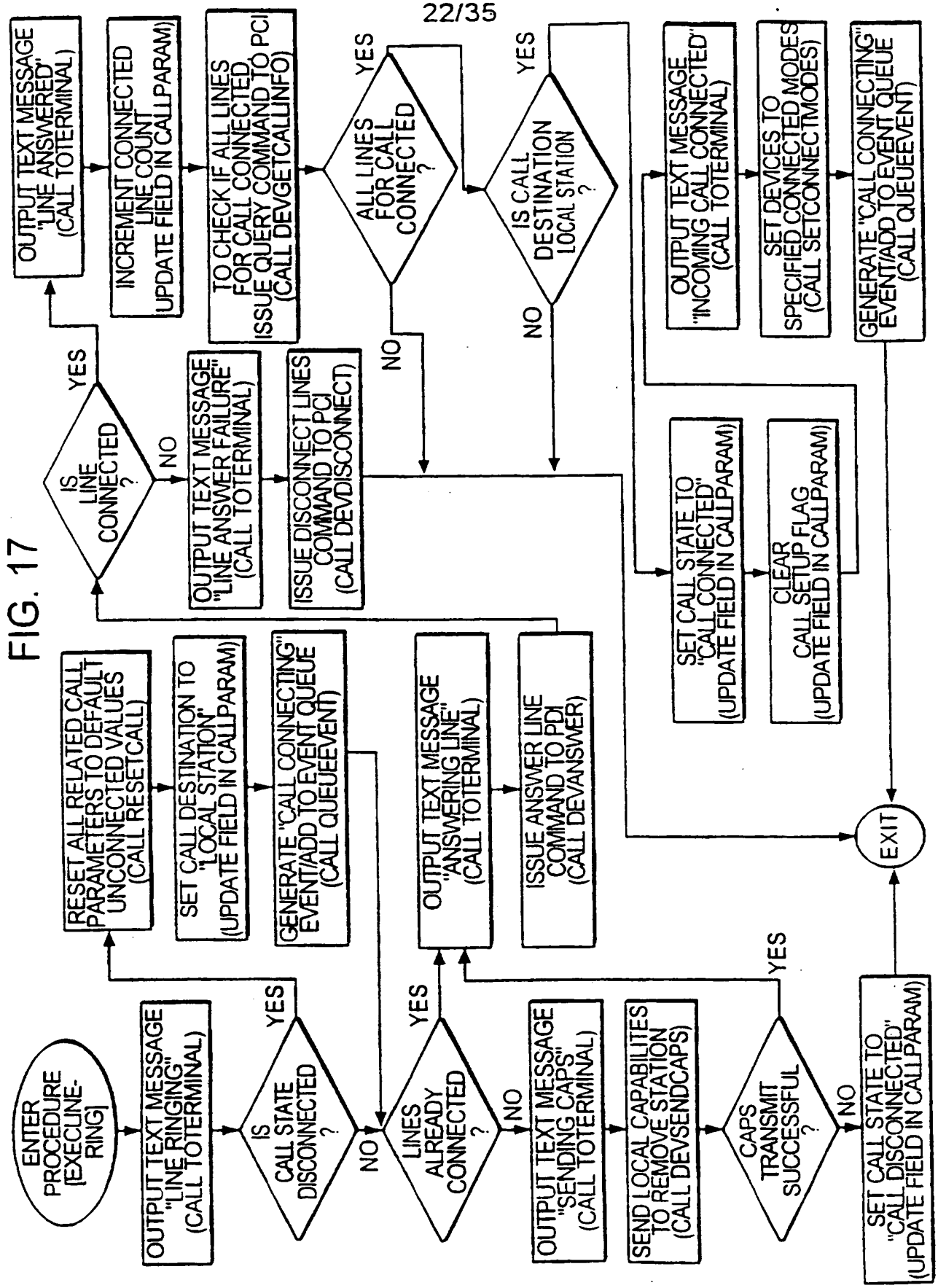


FIG. 17

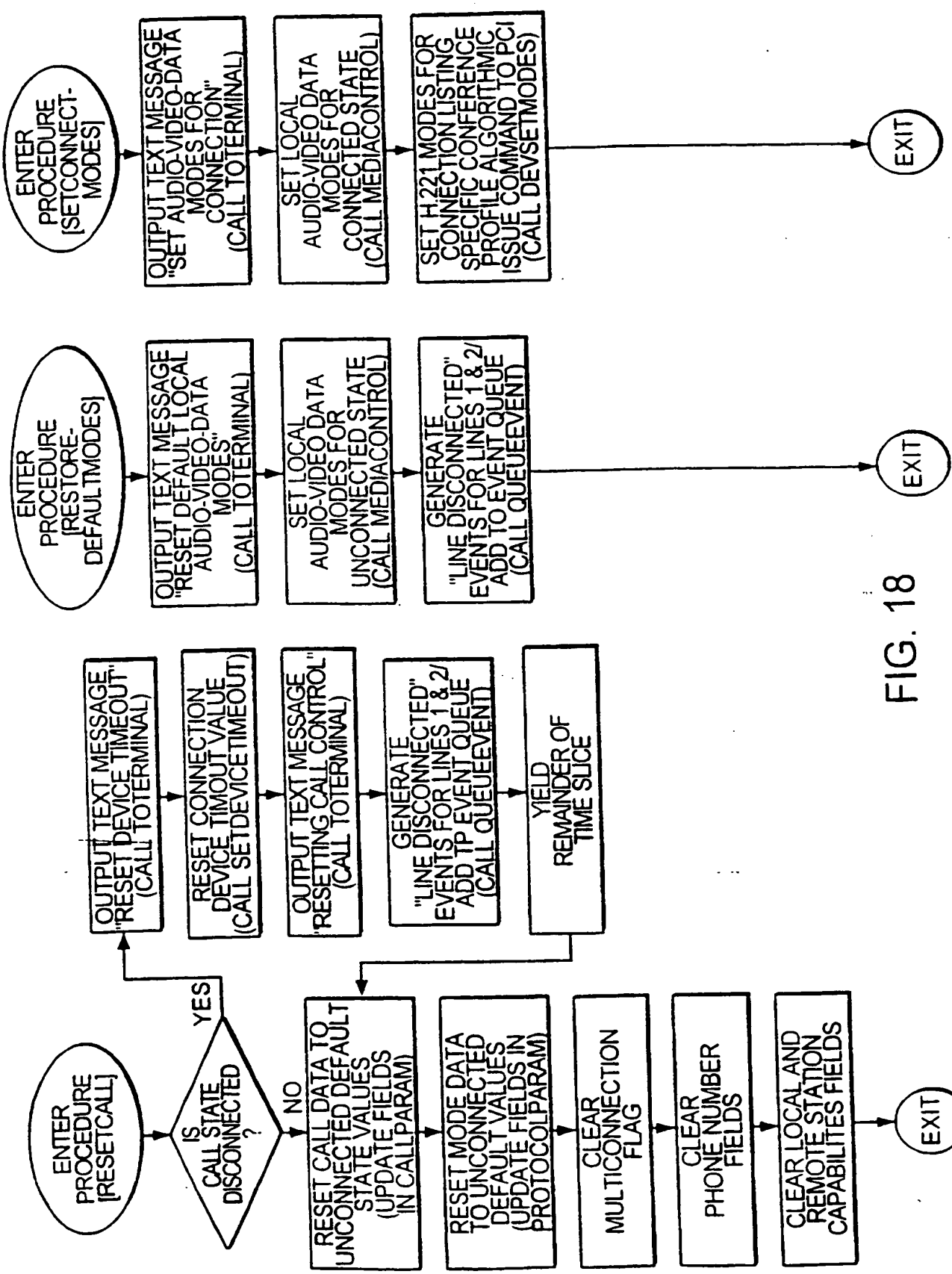

ENTER PROCEDURE [SETCONNECT-MODES]
OUTPUT TEXT MESSAGE "SET AUDIO-VIDEO-DATA MODES FOR CONNECTION" (CALL TOTERMINAL)
SET LOCAL AUDIO-VIDEO DATA MODES FOR CONNECTED STATE (CALL MEDIACONTROL)
SET H.221 MODES FOR CONNECTION LISTING SPECIFIC CONFERENCE PROFILE ALGORITHMIC ISSUE COMMAND TO PCI (CALL DEVSETMODES)
EXIT
ENTER PROCEDURE [RESTORE-DEFAULTMODES]
OUTPUT TEXT MESSAGE "RESET DEFAULT LOCAL AUDIO-VIDEO-DATA MODES" (CALL TOTERMINAL)
SET LOCAL AUDIO-VIDEO DATA MODES FOR UNCONNECTED STATE (CALL MEDIACONTROL)
GENERATE "LINE DISCONNECTED" EVENTS FOR LINES 1 & 2/ ADD TO EVENT QUEUE (CALL QUEUEEVENT)
EXIT
ENTER PROCEDURE [RESETCALL]
IS CALL STATE DISCONNECTED ?
YES
NO
OUTPUT TEXT MESSAGE "RESET DEVICE TIMOUT" (CALL TOTERMINAL)
RESET CONNECTION DEVICE TIMOUT VALUE (CALL SETDEVICETIMEOUT)
OUTPUT TEXT MESSAGE "RESETTING CALL CONTROL" (CALL TOTERMINAL)
GENERATE "LINE DISCONNECTED" EVENTS FOR LINES 1 & 2/ ADD TP EVENT QUEUE (CALL QUEUEEVENT)
YIELD REMAINDER OF TIME SLICE
RESET CALL DATA TO UNCONNECTED DEFAULT STATE VALUES (UPDATE FIELDS IN CALLPARAM)
RESET MODE DATA TO UNCONNECTED DEFAULT VALUES (UPDATE FIELDS IN PROTOCOLPARAM)
CLEAR MULTICONNECTION FLAG
CLEAR PHONE NUMBER FIELDS
CLEAR LOCAL AND REMOTE STATION CAPABILITES FIELDS
EXIT

FIG. 18

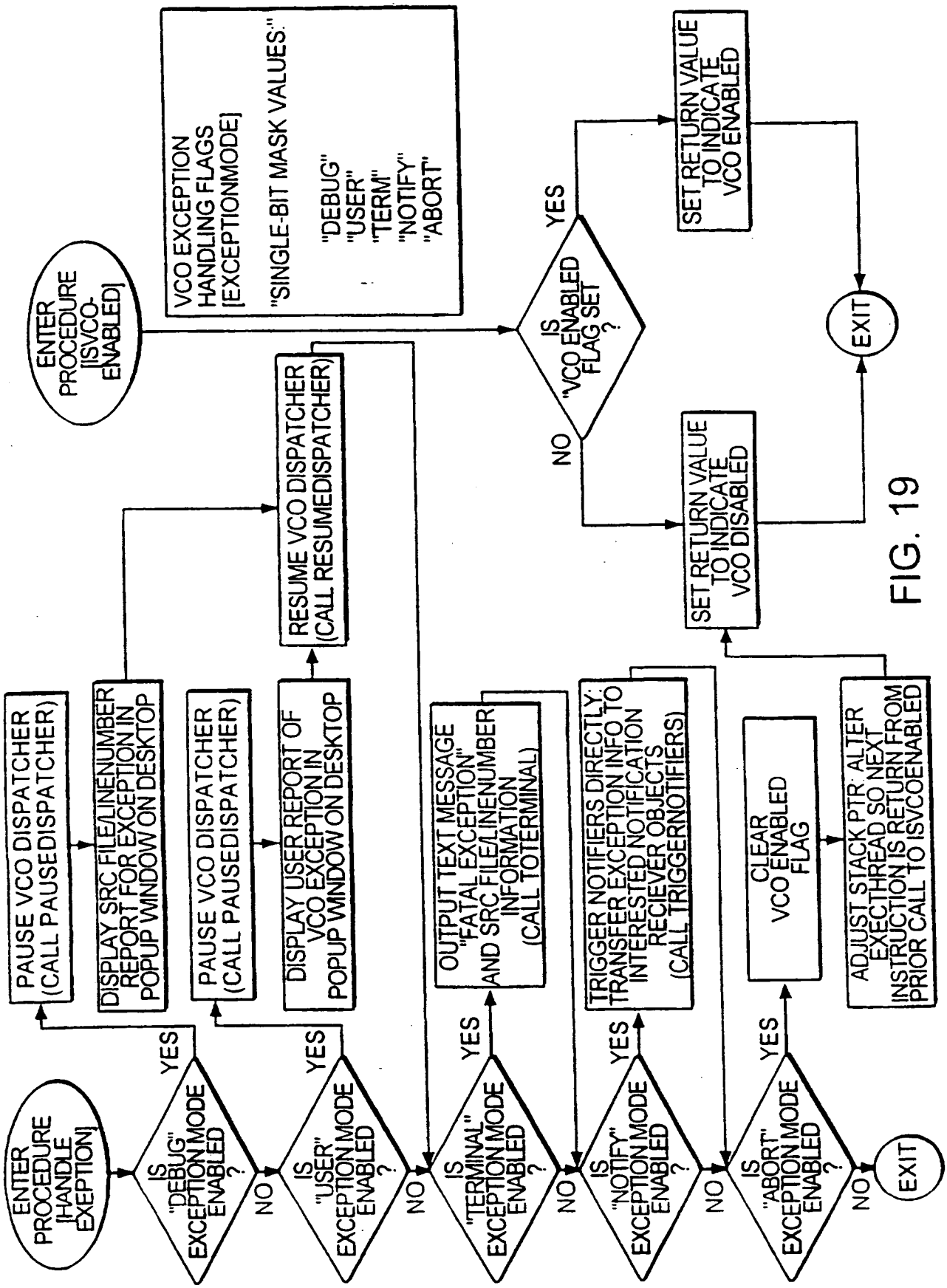
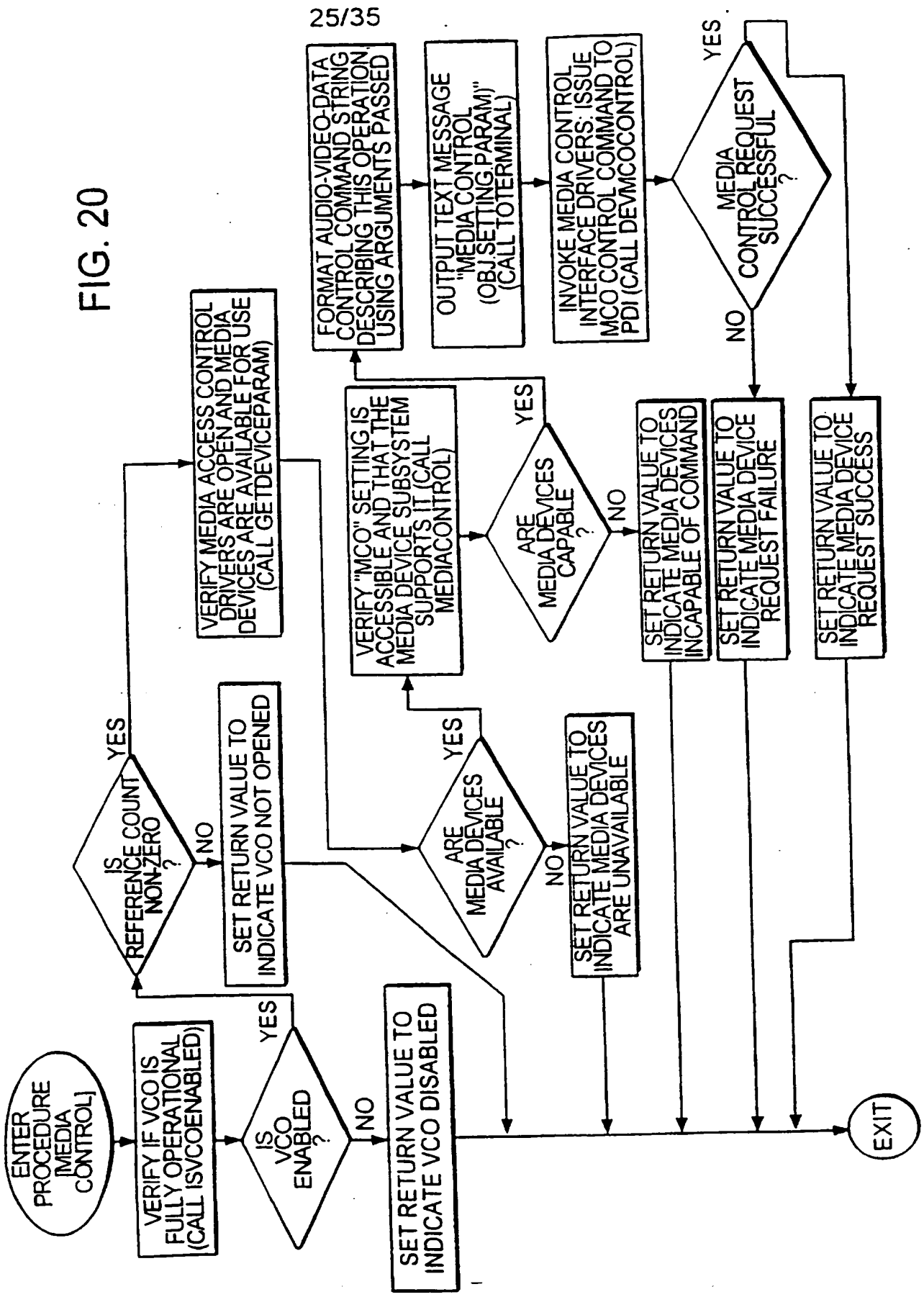


FIG. 19

FIG. 20


ENTER PROCEDURE MEDIA CONTROL
VERIFY IF VCO IS FULLY OPERATIONAL (CALL ISVCOENABLED)
IS VCO ENABLED ?
YES
NO
SET RETURN VALUE TO INDICATE VCO DISABLED
IS REFERENCE COUNT NON-ZERO ?
YES
NO
SET RETURN VALUE TO INDICATE VCO NOT OPENED
VERIFY MEDIA ACCESS CONTROL DRIVERS ARE OPEN AND MEDIA DEVICES ARE AVAILABLE FOR USE (CALL GETDEVICEPARAM)
ARE MEDIA DEVICES AVAILABLE ?
YES
NO
SET RETURN VALUE TO INDICATE MEDIA DEVICES ARE UNAVAILABLE
VERIFY "MCO" SETTING IS ACCESSIBLE AND THAT THE MEDIA DEVICE SUBSYSTEM SUPPORTS IT (CALL MEDIACONTROL)
ARE MEDIA DEVICES CAPABLE ?
YES
NO
SET RETURN VALUE TO INDICATE MEDIA DEVICES INCAPABLE OF COMMAND
FORMAT AUDIO-VIDEO-DATA CONTROL COMMAND STRING DESCRIBING THIS OPERATION USING ARGUMENTS PASSED
OUTPUT TEXT MESSAGE "MEDIA CONTROL (OBJ,SETTING,PARAM)" (CALL TOTERMINAL)
INVOKE MEDIA CONTROL INTERFACE DRIVERS: ISSUE MCO CONTROL COMMAND TO PDI (CALL DEVMCOCONTROL)
MEDIA CONTROL REQUEST SUCCESSFUL ?
YES
NO
SET RETURN VALUE TO INDICATE MEDIA DEVICE REQUEST FAILURE
SET RETURN VALUE TO INDICATE MEDIA DEVICE REQUEST SUCCESS
EXIT

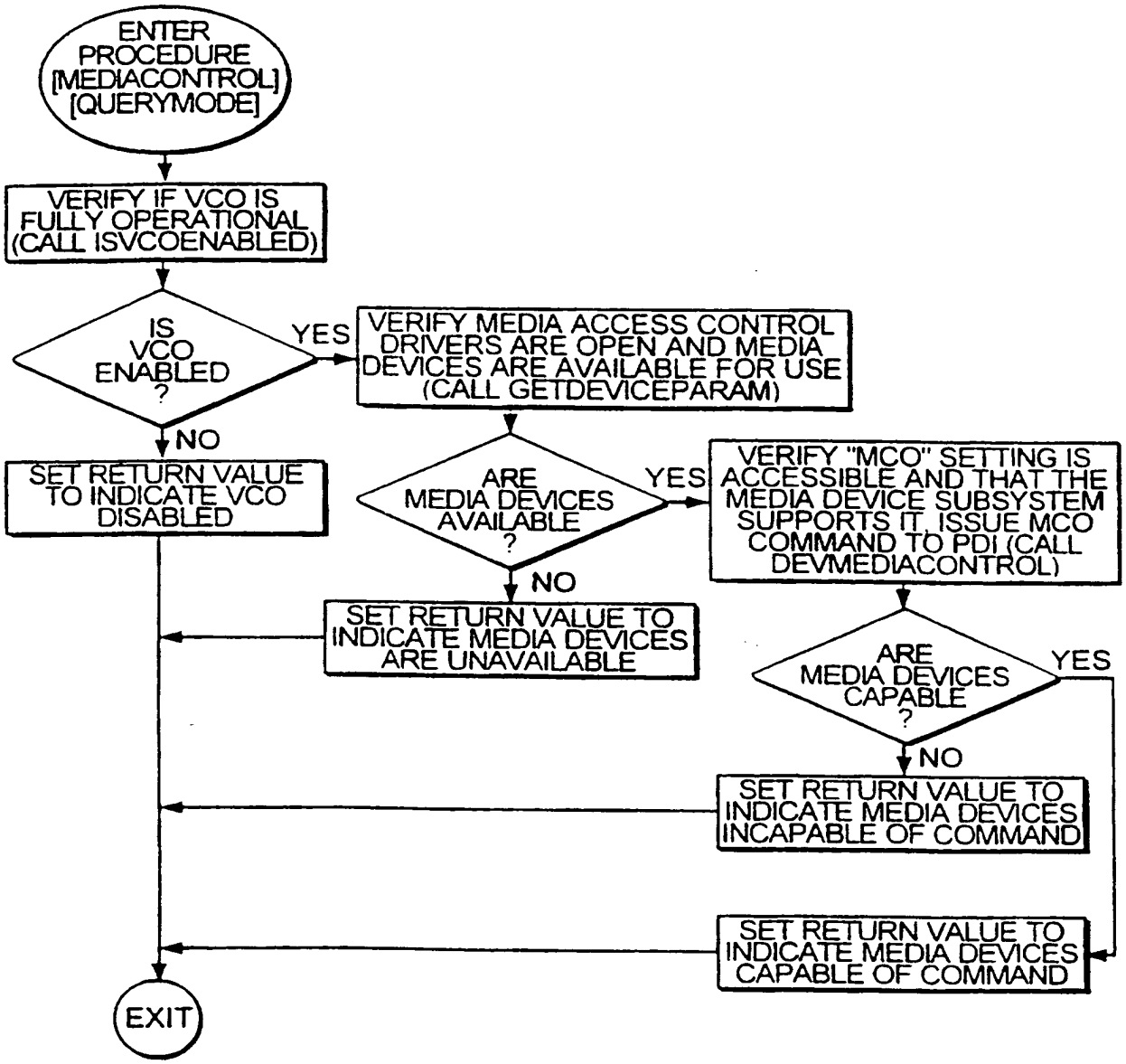

ENTER PROCEDURE [MEDIACONTROL] [QUERYMODE]
VERIFY IF VCO IS FULLY OPERATIONAL (CALL ISVCOENABLED)
IS VCO ENABLED ?
YES
NO
VERIFY MEDIA ACCESS CONTROL DRIVERS ARE OPEN AND MEDIA DEVICES ARE AVAILABLE FOR USE (CALL GETDEVICEPARAM)
SET RETURN VALUE TO INDICATE VCO DISABLED
ARE MEDIA DEVICES AVAILABLE ?
YES
NO
VERIFY "MCO" SETTING IS ACCESSIBLE AND THAT THE MEDIA DEVICE SUBSYSTEM SUPPORTS IT, ISSUE MCO COMMAND TO PDI (CALL DEVMEDIACONTROL)
SET RETURN VALUE TO INDICATE MEDIA DEVICES ARE UNAVAILABLE
ARE MEDIA DEVICES CAPABLE ?
YES
NO
SET RETURN VALUE TO INDICATE MEDIA DEVICES INCAPABLE OF COMMAND
SET RETURN VALUE TO INDICATE MEDIA DEVICES CAPABLE OF COMMAND
EXIT

FIG. 21

FIG. 22

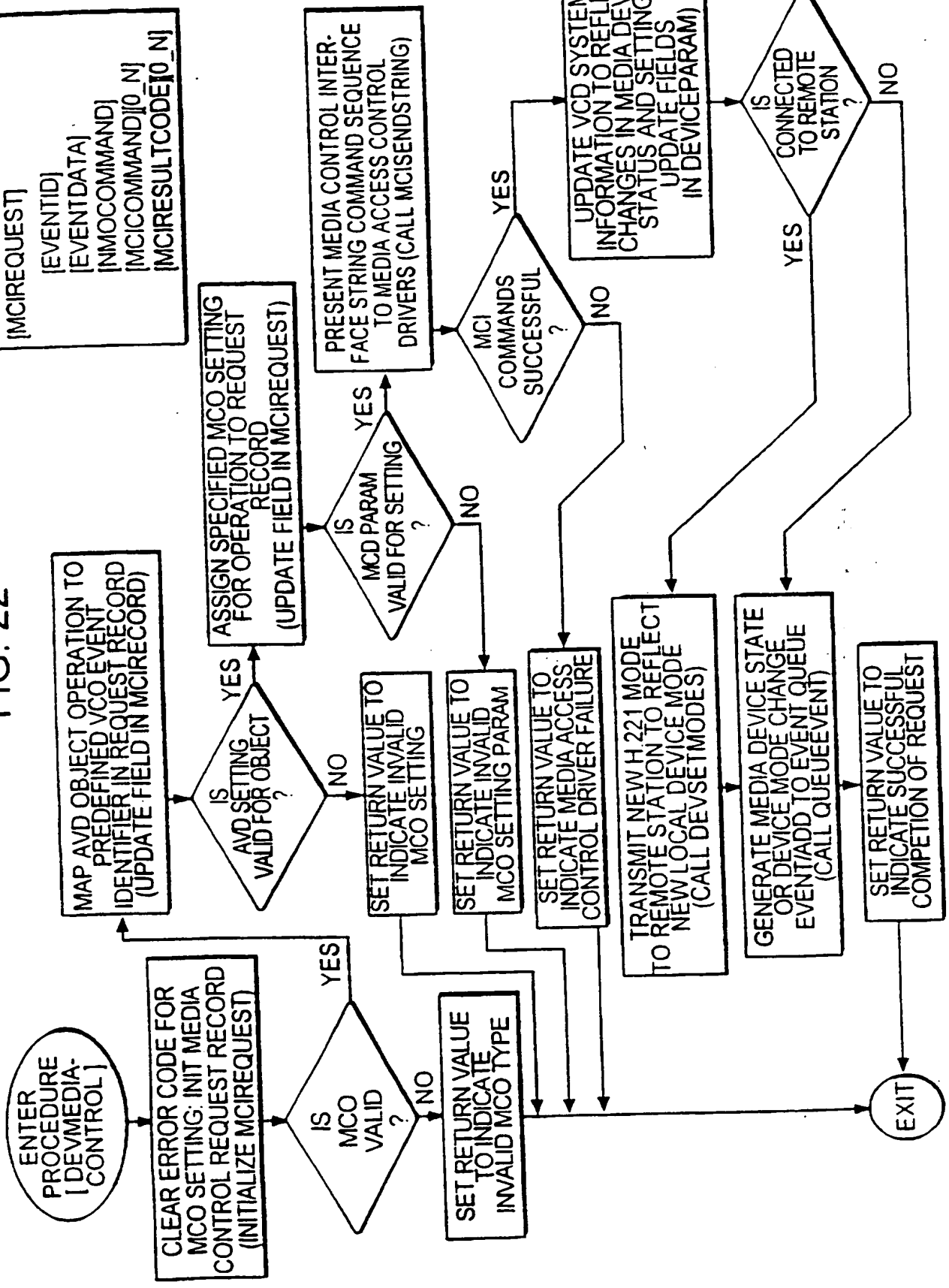
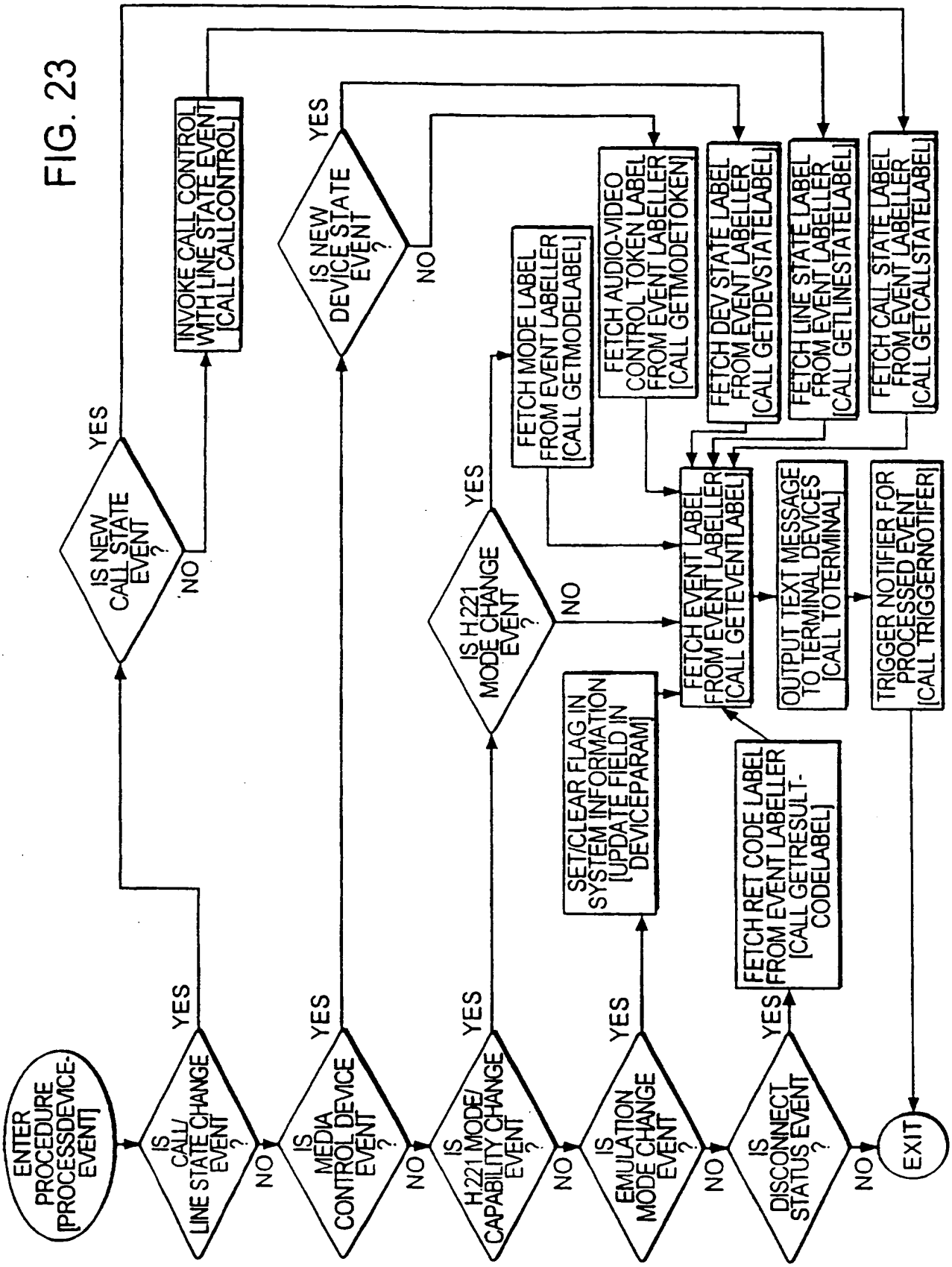

ENTER PROCEDURE [DEVMEDIA-CONTROL]
CLEAR ERROR CODE FOR MCO SETTING: INIT MEDIA CONTROL REQUEST RECORD (INITIALIZE MCIREQUEST)
IS MCO VALID ?
YES
NO
SET RETURN VALUE TO INDICATE INVALID MCO TYPE
MAP AVD OBJECT OPERATION TO PREDEFINED VCO EVENT IDENTIFIER IN REQUEST RECORD (UPDATE FIELD IN MCIREQUEST)
IS AVD SETTING VALID FOR OBJECT ?
YES
NO
SET RETURN VALUE TO INDICATE INVALID MCO SETTING
ASSIGN SPECIFIED MCO SETTING FOR OPERATION TO REQUEST RECORD (UPDATE FIELD IN MCIREQUEST)
IS MCD PARAM VALID FOR SETTING ?
YES
NO
SET RETURN VALUE TO INDICATE INVALID MCO SETTING PARAM
PRESENT MEDIA CONTROL INTER-FACE STRING COMMAND SEQUENCE TO MEDIA ACCESS CONTROL DRIVERS (CALL MCISENDSTRING)
MCI COMMANDS SUCCESSFUL ?
YES
NO
SET RETURN VALUE TO INDICATE MEDIA ACCESS CONTROL DRIVER FAILURE
UPDATE VCD SYSTEM INFORMATION TO REFLECT CHANGES IN MEDIA DEVICE STATUS AND SETTINGS UPDATE FIELDS IN DEVICEPARAM)
IS CONNECTED TO REMOTE STATION ?
YES
NO
TRANSMIT NEW H.221 MODE TO REMOTE STATION TO REFLECT NEW LOCAL DEVICE MODE (CALL DEVSETMODES)
GENERATE MEDIA DEVICE STATE OR DEVICE MODE CHANGE EVENT/ADD TO EVENT QUEUE (CALL QUEUEEVENT)
SET RETURN VALUE TO INDICATE SUCCESSFUL COMPETION OF REQUEST
EXIT
MCO SETTING REQUEST [MCIREQUEST] [EVENTID] [EVENTDATA] [NMOCOMMAND] [MCICOMMAND][0_N] [MCIRESULTCODE][0_N]

FIG. 23



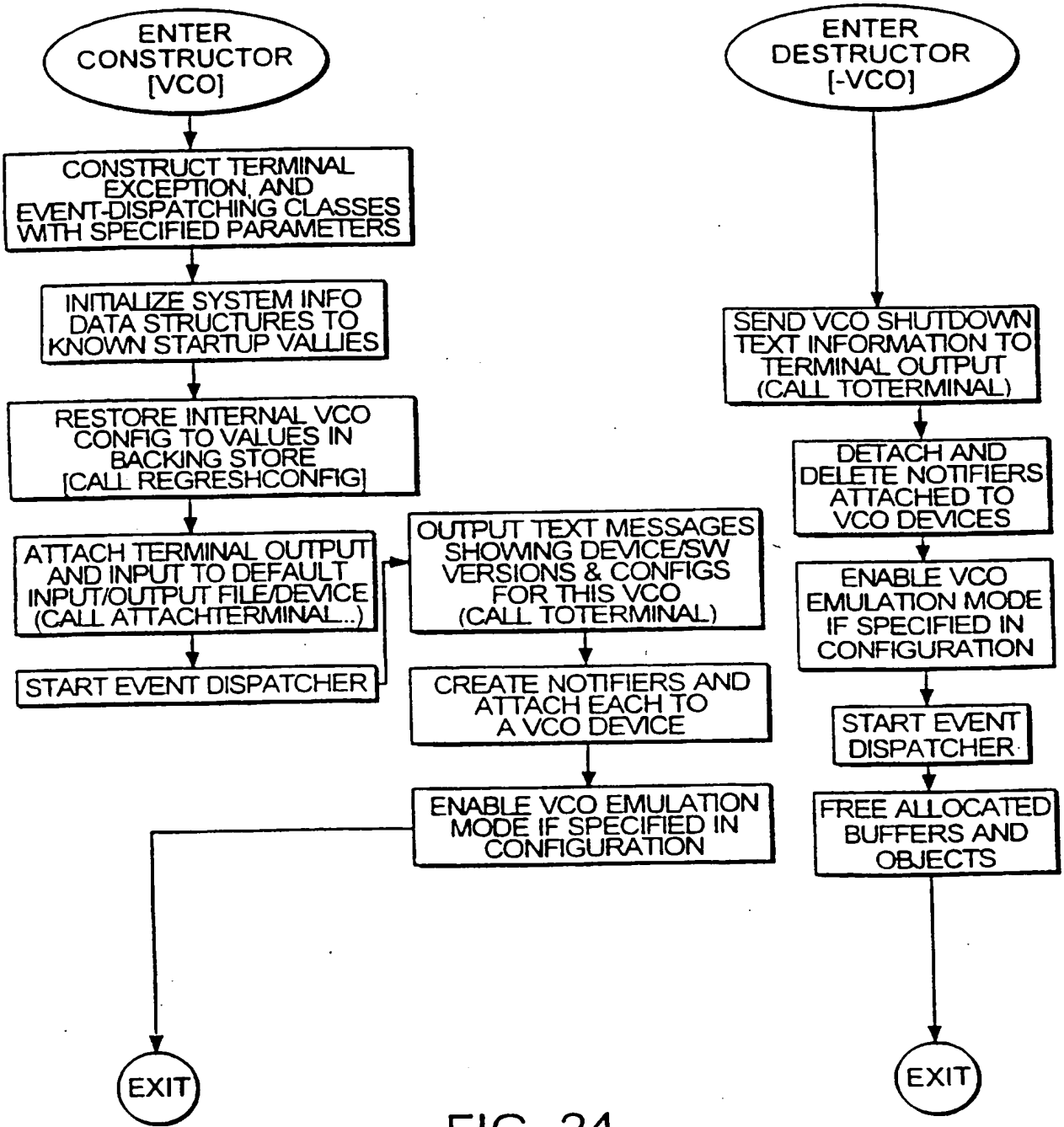

ENTER CONSTRUCTOR [VCO]
CONSTRUCT TERMINAL EXCEPTION, AND EVENT-DISPATCHING CLASSES WITH SPECIFIED PARAMETERS
INITIALIZE SYSTEM INFO DATA STRUCTURES TO KNOWN STARTUP VALUES
RESTORE INTERNAL VCO CONFIG TO VALUES IN BACKING STORE [CALL REGRESHCONFIG]
ATTACH TERMINAL OUTPUT AND INPUT TO DEFAULT INPUT/OUTPUT FILE/DEVICE (CALL ATTACHTERMINAL...)
START EVENT DISPATCHER
OUTPUT TEXT MESSAGES SHOWING DEVICE/SW VERSIONS & CONFIGS FOR THIS VCO (CALL TOTERMINAL)
CREATE NOTIFIERS AND ATTACH EACH TO A VCO DEVICE
ENABLE VCO EMULATION MODE IF SPECIFIED IN CONFIGURATION
EXIT
ENTER DESTRUCTOR [-VCO]
SEND VCO SHUTDOWN TEXT INFORMATION TO TERMINAL OUTPUT (CALL TOTERMINAL)
DETACH AND DELETE NOTIFIERS ATTACHED TO VCO DEVICES
ENABLE VCO EMULATION MODE IF SPECIFIED IN CONFIGURATION
START EVENT DISPATCHER
FREE ALLOCATED BUFFERS AND OBJECTS
EXIT

FIG. 24

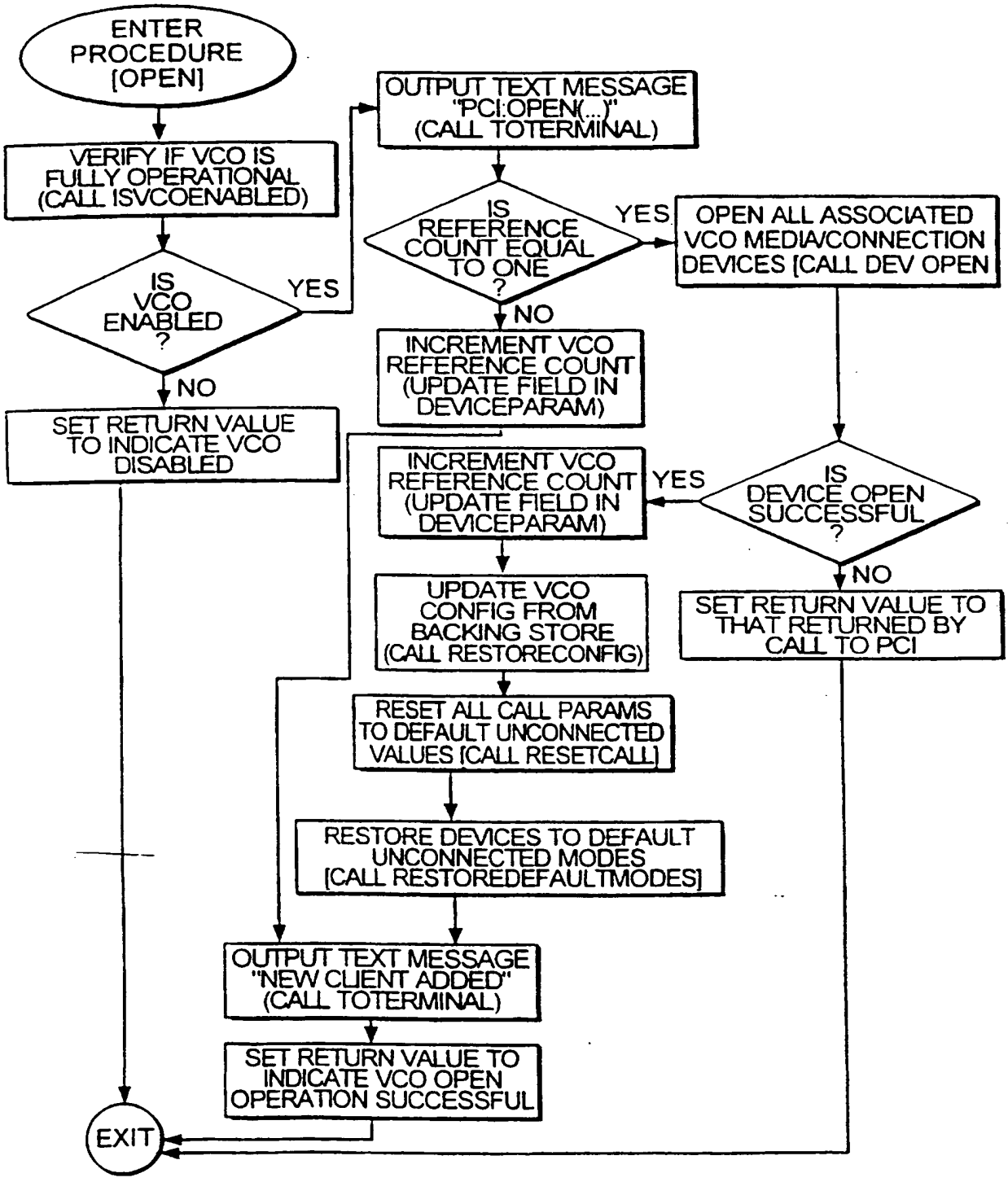


FIG. 25

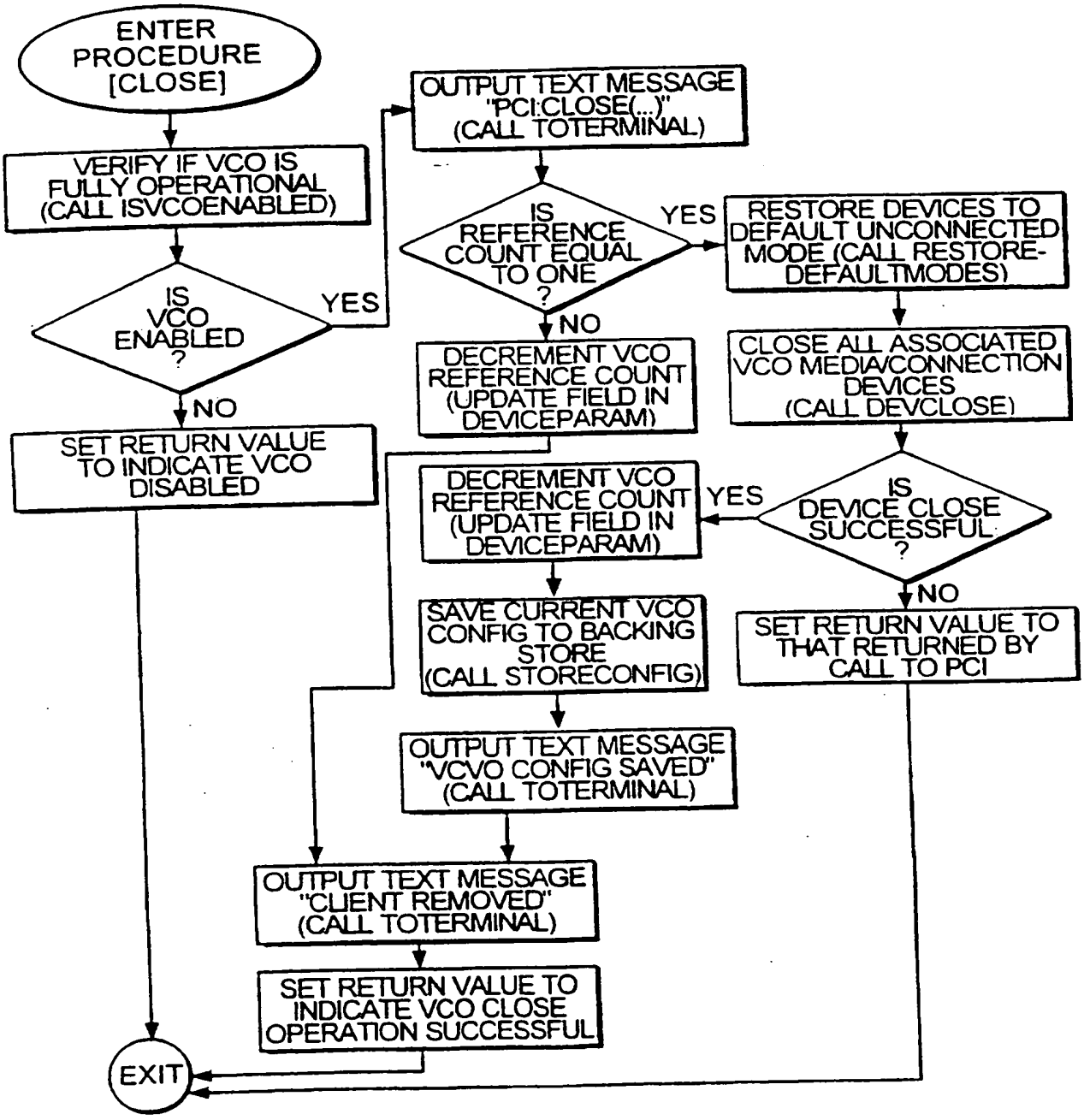
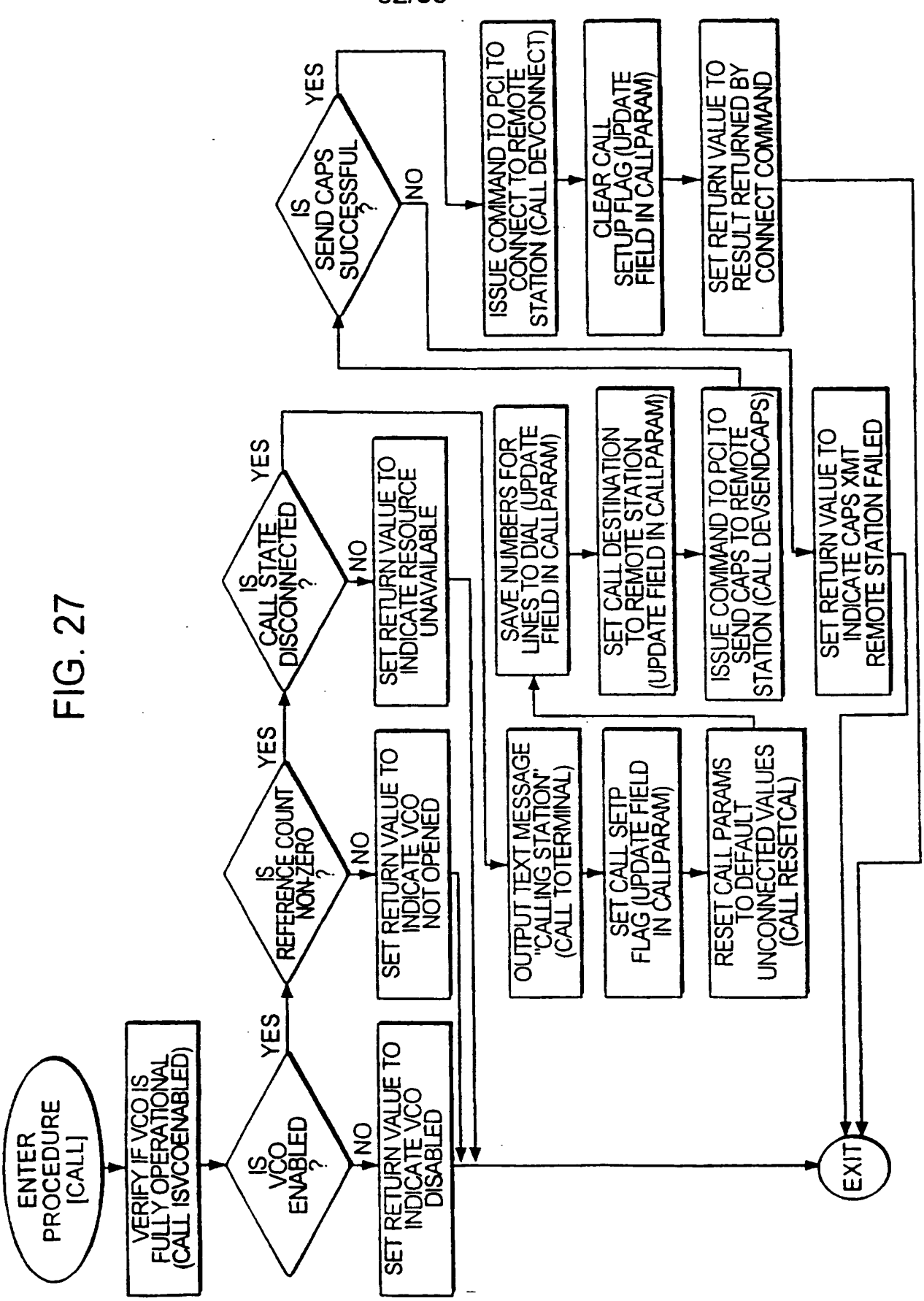

ENTER PROCEDURE [CLOSE]
VERIFY IF VCO IS FULLY OPERATIONAL (CALL ISVCOENABLED)
IS VCO ENABLED ?
YES
NO
SET RETURN VALUE TO INDICATE VCO DISABLED
OUTPUT TEXT MESSAGE "PCI:CLOSE(...)" (CALL TOTERMINAL)
IS REFERENCE COUNT EQUAL TO ONE ?
YES
NO
RESTORE DEVICES TO DEFAULT UNCONNECTED MODE (CALL RESTORE-DEFAULTMODES)
DECREMENT VCO REFERENCE COUNT (UPDATE FIELD IN DEVICEPARAM)
CLOSE ALL ASSOCIATED VCO MEDIA/CONNECTION DEVICES (CALL DEVCLOSE)
DECREMENT VCO REFERENCE COUNT (UPDATE FIELD IN DEVICEPARAM)
YES
IS DEVICE CLOSE SUCCESSFUL ?
NO
SAVE CURRENT VCO CONFIG TO BACKING STORE (CALL STORECONFIG)
SET RETURN VALUE TO THAT RETURNED BY CALL TO PCI
OUTPUT TEXT MESSAGE "VCVO CONFIG SAVED" (CALL TOTERMINAL)
OUTPUT TEXT MESSAGE "CLIENT REMOVED" (CALL TOTERMINAL)
SET RETURN VALUE TO INDICATE VCO CLOSE OPERATION SUCCESSFUL
EXIT

FIG. 26

FIG. 27



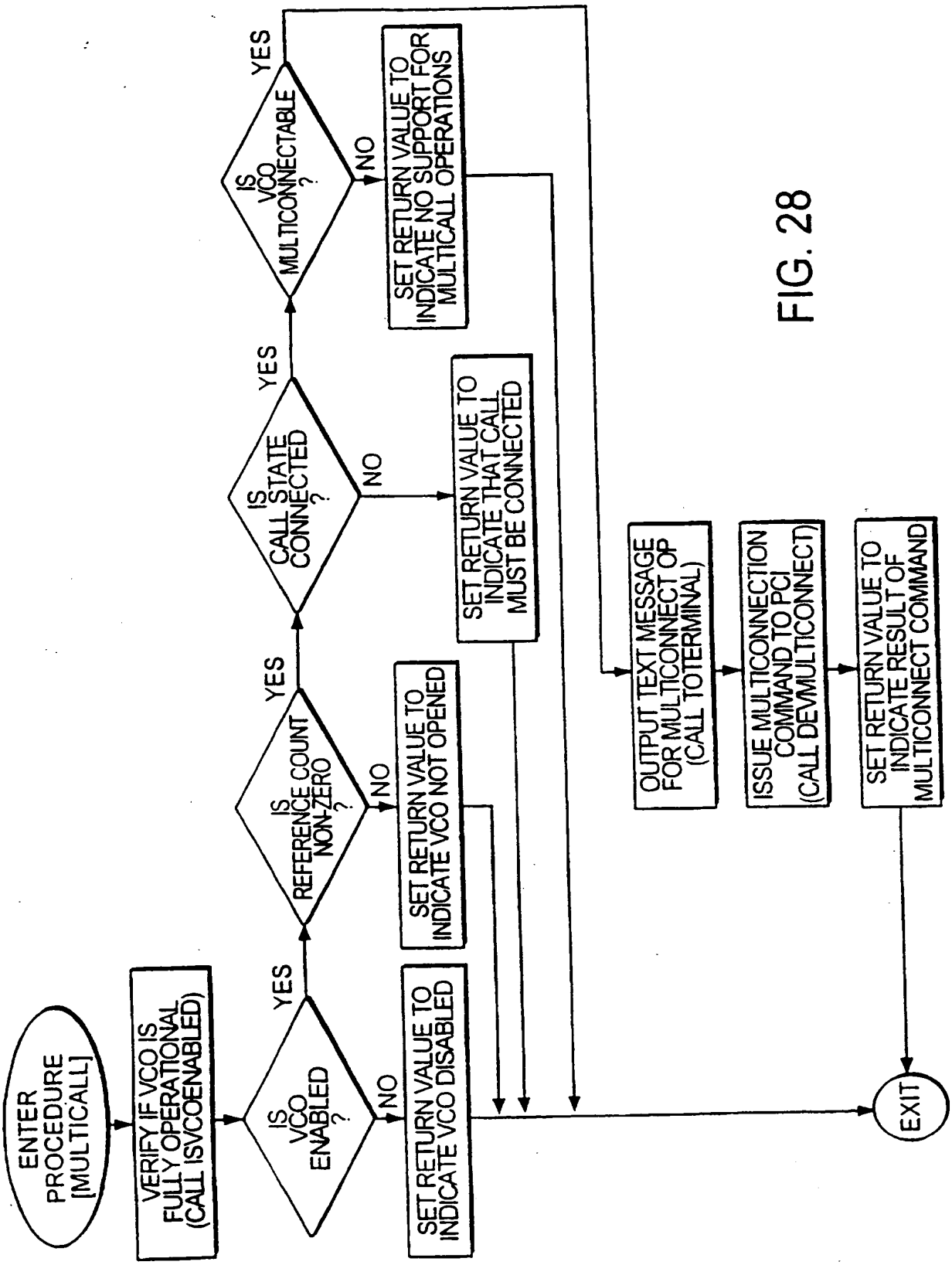

ENTER PROCEDURE [MULTICALL]
VERIFY IF VCO IS FULLY OPERATIONAL (CALL ISVCOENABLED)
IS VCO ENABLED ?
YES
NO
SET RETURN VALUE TO INDICATE VCO DISABLED
IS REFERENCE COUNT NON-ZERO ?
YES
NO
SET RETURN VALUE TO INDICATE VCO NOT OPENED
IS CALL STATE CONNECTED ?
YES
NO
SET RETURN VALUE TO INDICATE THAT CALL MUST BE CONNECTED
IS VCO MULTICONNECTABLE ?
YES
NO
SET RETURN VALUE TO INDICATE NO SUPPORT FOR MULTICALL OPERATIONS
OUTPUT TEXT MESSAGE FOR MULTICONNECT OP (CALL TOTERMINAL)
ISSUE MULTICONNECTION COMMAND TO PCI (CALL DEVMULTICONNECT)
SET RETURN VALUE TO INDICATE RESULT OF MULTICONNECT COMMAND
EXIT

FIG. 28

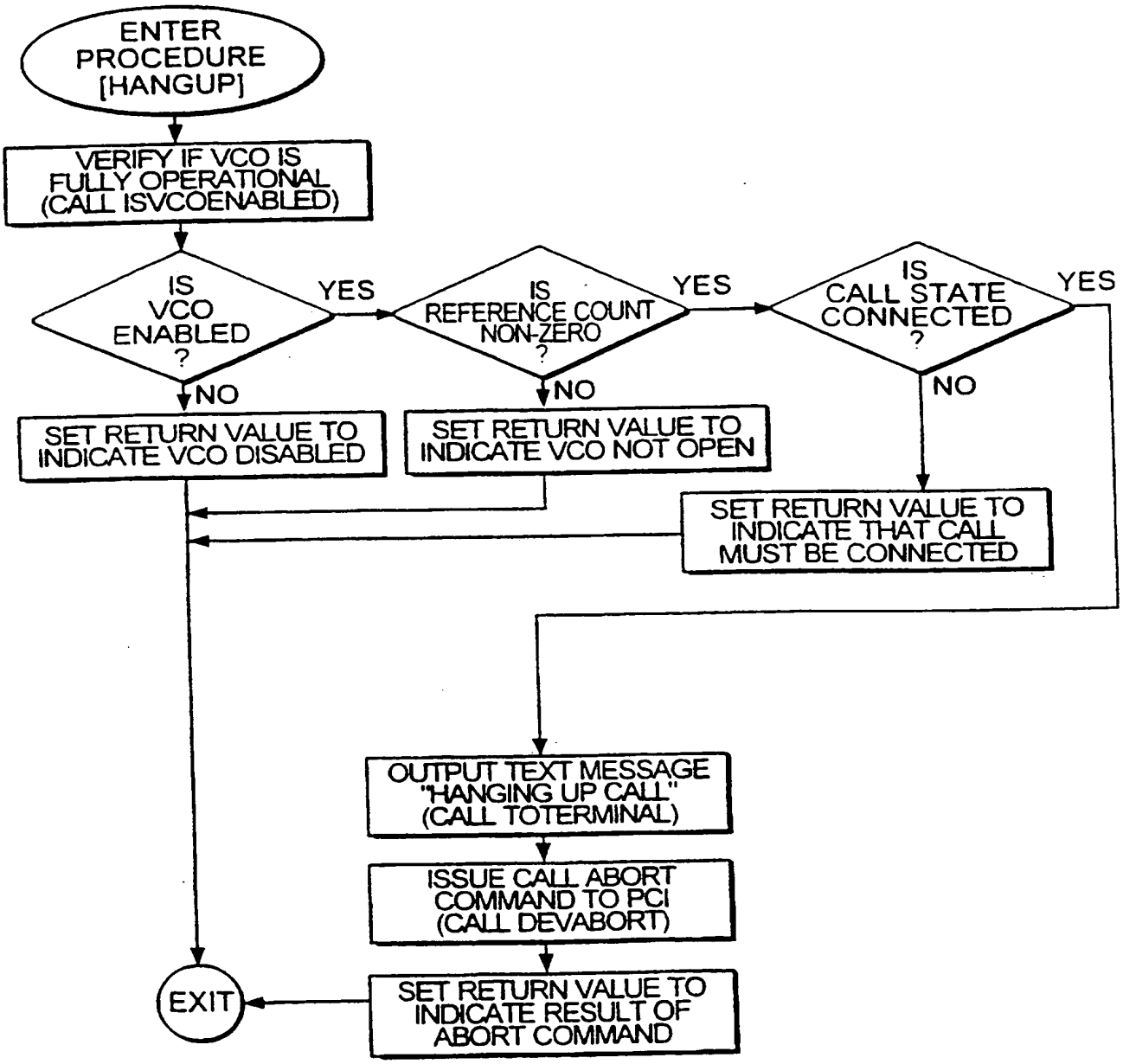

ENTER PROCEDURE [HANGUP]
VERIFY IF VCO IS FULLY OPERATIONAL (CALL ISVCOENABLED)
IS VCO ENABLED ?
YES
NO
SET RETURN VALUE TO INDICATE VCO DISABLED
IS REFERENCE COUNT NON-ZERO ?
YES
NO
SET RETURN VALUE TO INDICATE VCO NOT OPEN
IS CALL STATE CONNECTED ?
YES
NO
SET RETURN VALUE TO INDICATE THAT CALL MUST BE CONNECTED
OUTPUT TEXT MESSAGE "HANGING UP CALL" (CALL TOTERMINAL)
ISSUE CALL ABORT COMMAND TO PCI (CALL DEVABORT)
SET RETURN VALUE TO INDICATE RESULT OF ABORT COMMAND
EXIT

FIG. 29

OPERATION	DESCRIPTION
SetConFocus	Set conference focus to specified station
QueryConfFocus	Determine identity of station currently in focus
SetConfChair	Set conference chairman
QueryConfChair	Determine identity of conference chairmen
AddStation	Add station to conference
RemoveStation	Remove station from conference
BroadcastAudio	Enable or disable broadcasting of audio conference
BroadcastVideo	Enable or disable broadcasting of video conference
BroadcastData	Enable or disable broadcasting of data conference
GetNumStations	Get number of conferees
GetStationList	Get list of conferees
GetStationCaps	Get capabilites of particular conferee
GetStationAudio	Get audio of particular conferee
GetStationVideo	Get video of particular conferee
GetStationData	Get data of particular conferee
GetStationIdentity	Get numbers and station label of particular conferee

MULTIPOINT CONTROL OPERATIONS

FIG. 30

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US97/15018

A. CLASSIFICATION OF SUBJECT MATTER

IPC(6) :G06F 9/06

US CL :395/182.02, 200.04, 200.02, 500, 651, 653; 370/463

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

U.S. : 395/182.02, 200.04, 200.02, 500, 651, 653; 370/463

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

IEEE, APS

search terms: physical layer, virtual layer, multimedia, virtual interface, control interface, logical function, relational

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	US 5,226,160 A (WALDRON ET AL.) 06 July 1993, Col. 3, lines 48-57, Col. 5, lines 45-66, Col. 7, lines 45-55, Col. 8, lines 21-45.	1-11
Y	US 4,677,588 A (BENJAMIN et al.) 30 June 1987, cols. 4-10.	1-11
Y	US 5,483,647 A (YU et al.) 09 January 1996, entire document.	1-11
A	US 5,138,614 A (BAUMGARTNER et al.) 11 August 1992.	1-11

☐ Further documents are listed in the continuation of Box C. ☐ See patent family annex.

*	Special categories of cited documents:	"T"	later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"A"	document defining the general state of the art which is not considered to be of particular relevance	"X"	document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"E"	earlier document published on or after the international filing date	"Y"	document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art
"L"	document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)	"&"	document member of the same patent family
"O"	document referring to an oral disclosure, use, exhibition or other means		
"P"	document published prior to the international filing date but later than the priority date claimed		

Date of the actual completion of the international search 24 OCTOBER 1997	Date of mailing of the international search report 22 DEC 1997
Name and mailing address of the ISA/US Commissioner of Patents and Trademarks Box PCT Washington, D.C. 20231 Facsimile No. (703) 305-3230	Authorized officer THAI PHAN Telephone No. (703) 305-9704

Form PCT/ISA/210 (second sheet)(July 1992)*